Verifying C Programs: A VCC Tutorial

Working draft, version 0.2, July 10, 2015

Ernie Cohen, Mark A. Hillebrand, Stephan Tobies
European Microsoft Innovation Center
{ecohen,mahilleb,stobies}@microsoft.com

Michał Moskal, Wolfram Schulte
Microsoft Research Redmond
{micmo,schulte}@microsoft.com

Abstract

VCC is a verification environment for software written in C. VCC takes a program (annotated with function contracts, state assertions, and type invariants) and attempts to prove that these annotations are correct, i.e. that they hold for every possible program execution. The environment includes tools for monitoring proof attempts and constructing partial counterexample executions for failed proofs. VCC handles fine-grained concurrency and low-level C features, and has been used to verify the functional correctness of tens of thousands of lines of commercial concurrent system code.

This tutorial describes how to use VCC to verify C code. It covers the annotation language, the verification methodology, and the use of VCC itself.

1. Introduction

This tutorial is an introduction to verifying C code with VCC. Our primary audience is programmers who want to write correct code. The only prerequisite is a working knowledge of C.

To use VCC, you first *annotate* your code to specify what your code does (e.g., that your sorting function sorts its input), and why it works (e.g., suitable invariants for your loops and data structures). VCC then tries to *prove* (mathematically) that your program meets these specifications. Unlike most program analyzers, VCC doesn't look for bugs, or analyze an abstraction of your program; if VCC certifies that your program is correct, then your program really is correct[1].

To check your program, VCC uses the *deductive verification* paradigm: it generates a number of mathematical statements (called ***verification conditions***), the validity of which suffice to guarantee the program's correctness, and tries to prove these statements using an automatic theorem prover. If any of these proofs fail, VCC reflects these failures back to you in terms of the program itself (as opposed to the formulas seen by the theorem prover). Thus, you normally interact with VCC entirely at the level of code and program states; you can usually ignore the mathematical reasoning going on "under the hood". For example, if your program uses division somewhere, and VCC is unable to prove that the divisor is nonzero, it will report this to you as a (potential) program error at that point in the program. This doesn't mean that your program is necessarily incorrect; most of the time, especially when verifying code that is already well-tested, it is because you haven't provided enough information to allow VCC to deduce that the suspected error doesn't occur. (For example, you might have failed to specify that some function parameter is required to be nonzero.) Typically, you fix this "error" by strengthening your annotations. This might lead to other error reports, forcing you to add additional annotations, so verification is in practice an iterative process. Sometimes, this process will reveal a genuine programming error. But even if it doesn't, you will have not only proved your code free from such errors, but you will have produced the precise specifications for your code – a very useful form of documentation.

This tutorial covers basics of VCC annotation language. By the time you have finished working through it, you should be able to use VCC to verify some nontrivial programs. It doesn't cover the theoretical background of VCC [2], implementation details [1] or advanced topics; information on these can be found on the VCC homepage[2]. More information on all topics covered in the tutorial can be found in the VCC manual.

The examples in this tutorial are currently distributed with the VCC sources.[3]

You can use VCC either from the command line or from Visual Studio 2008/2010 (VS); the VS interface offers easy access to different components of VCC tool chain and is thus generally recommended. VCC can be downloaded from the VCC homepage; be sure to read the installation instructions[4], which provide important information about installation prerequisites and how to set up tool paths.

2. Verifying Simple Programs

We begin by describing how VCC verifies "simple programs" — sequential programs without loops, function calls, or concurrency. This might seem to be a trivial subset of C, but in fact VCC reasons about more complex programs by reducing them to reasoning about simple programs.

2.1 Assertions

Let's begin with a simple example:

```
#include <vcc.h>
```

[1] In reality, this isn't necessarily true, for two reasons. First, VCC itself might have bugs; in practice, these are unlikely to cause you to accidentally verify an incorrect program, unless you find and intentionally exploit such a bug. Second, there are a few checks needed for soundness that haven't yet been added to VCC, such as checking that ghost code terminates; these issues are listed in section § **??**.

[2] `http://vcc.codeplex.com/`

[3] Available from `http://vcc.codeplex.com/SourceControl/list/changesets`: click Download on the right, get the zip file and navigate to vcc/Docs/Tutorial/c.

[4] `http://vcc.codeplex.com/wikipage?title=Install`

```
int main()
{
  int x,y,z;
  if (x <= y)
    z = x;
  else z = y;
  _(assert z <= x)
  return 0;
}
```

This program sets z to the minimum of x and y. In addition to the ordinary C code, this program includes an ***annotation***, starting with _(, terminating with a closing parenthesis, with balanced parentheses inside. The first identifier after the opening parenthesis (in the program above it's assert) is called an ***annotation tag*** and identifies the type of annotation provided (and hence its function). (The tag plays this role only at the beginning of an annotation; elsewhere, it is treated as an ordinary program identifier.) Annotations are used only by VCC, and are not seen by the C compiler. When using the regular C compiler the <vcc.h> header file defines:

```
#define _(...) /* nothing */
```

VCC does not use this definition, and instead parses the inside of _(...) annotations.

An annotation of the form _(assert E), called an ***assertion***, asks VCC to prove that E is guaranteed to hold (i.e., evaluate to a value other than 0) whenever control reaches the assertion.[5] Thus, the line _(assert z <= x) says that when control reaches the assertion, z is no larger than x. If VCC successfully verifies a program, it promises that this will hold throughout every possible execution of the program, regardless of inputs, how concurrent threads are scheduled, etc. More generally, VCC might verify some, but not all of your assertions; in this case, VCC promises that the first assertion to be violated will not be one of the verified ones.

> It is instructive to compare _(assert E) with the macro assert(E) (defined in <assert.h>), which evaluates E at runtime and aborts execution if E doesn't hold. First, assert(E) requires runtime overhead (at least in builds where the check is made), whereas _(assert E) does not. Second, assert(E) will catch failure of the assertion only when it actually fails in an execution, whereas _(assert E) is guaranteed to catch the failure if it is possible in *any* execution. Third, because _(assert E) is not actually executed, E can include unimplementable mathematical operations, such as quantification over infinite domains.

To verify the function using VCC from the command line, save the source in a file called (say) minimum.c and run VCC as follows:

```
C:\Somewhere\VCC Tutorial> vcc.exe minimum.c
Verification of main succeeded.
C:\Somewhere\VCC Tutorial>
```

If instead you wish to use VCC Visual Studio plugin, load the solution tutorial.sln in <vcc source dir>\vcc\vcc\Docs\Tutorial, locate the file with the example, and right-click on the program text. You should get options to verify the file or just this function (either will work). The solution has the examples from this tutorial arrange by section, named so that they will appear within the project in textual order.

If you right-click within a C source file, several VCC commands are made available, depending on what kind of construction IntelliSense thinks you are in the middle of. The choice of verifying the entire file is always available. If you click within the definition of a struct type, VCC will offer you the choice of checking admissibility for that type (a concept explained in § 5.5). If you click within the body of a function, VCC should offer you the opportunity to verify just that function. However, IntelliSense often gets confused about the syntactic structure of VCC code, so it might not always present these context-dependent options. However, if you select the name of a function and then right click, it will allow you to verify just that function.

VCC verifies this function successfully, which means that its assertions are indeed guaranteed to hold and that the program cannot crash.[6] If VCC is unable to prove an assertion, it reports an error. Try changing the assertion in this program to something that isn't true and see what happens. (You might also want to try coming up with some complex assertion that is true, just to see whether VCC is smart enough to prove it.)

To understand how VCC works, and to use it successfully, it is useful to think in terms of what VCC "knows" at each control point of your program. In the current example, just before the first conditional, VCC knows nothing about the local variables, since they can initially hold any value. Just before the first assignment, VCC knows that x <= y (because execution followed that branch of the conditional), and after the assignment, VCC also knows that z == x, so it knows that z <= x. Similarly, in the else branch, VCC knows that y < x (because execution didn't follow the if branch), and after the assignment to z, it also knows that z == y, so it also knows z <= x. Since z <= x is known to hold at the end of each branch of the conditional, it is known to hold at the end of the conditional, so the assertion is known to hold when control reaches it. In general, VCC doesn't lose any information when reasoning about assignments and conditionals. However, we will see that VCC may lose some information when reasoning about loops; you will then need to provide annotations to make sure that it doesn't lose "too much".

When we talk about what VCC knows, we mean what it knows in an ideal sense, where if it knows E, it also knows any logical consequence of E. In such a world, adding an assertion that succeeds would have no effect on whether later assertions succeed. VCC's ability to deduce consequences is indeed complete for many types of formulas (e.g. formulas that use only equalities, inequalities, addition, subtraction, multiplication by constants, and boolean connectives), but not for all formulas, so VCC will sometimes fail to prove an assertion, even though it "knows" enough to prove it. Conversely, an assertion that succeeds can sometimes cause later assertions that would otherwise fail to succeed, by drawing VCC's attention to a crucial fact it needs to prove the later assertion. This is relatively rare, and typically involves "nonlinear arithmetic" (e.g. where variables are multiplied together), bitvector reasoning (§ 2.4.1) or quantifiers.

When VCC surprises you by failing to verify something that you think it should be able to verify, it is usually because it doesn't know something you think it should know. An assertion provides one way to check whether VCC knows what you think it knows.

Exercises

1. Can the assertion at the end of the example function be made stronger? What is the strongest valid assertion one could write? Try verifying the program with your stronger assertion.
2. Write an assertion that says that the int x is the average of the ints y and z.

[5] This interpretation changes slightly if E refers to memory locations that might be concurrently modified by other threads; see § 8

[6] VCC will not check that this program terminates, because the program hasn't been specified as having to terminate; we will see later how to provide such specifications. VCC also doesn't check that the program doesn't run out of stack space, but this feature may be provided in some future version.

3. Modify the example program so that it sets x and y to values that differ by at most 1 and sum to z. Prove that your program does this correctly.

Solutions

1. The strongest postcondition is _(assert z<=x && z<=y && (z == x || z == y)).
2. _(assert 2 * x == y + z). (Note that _(x == (y + z)/2) doesn't quite do the job; you might get a false positive because division rounds down.)
3.
```
int main() {
  int x,y,z;
  x = z/2;
  y = z-x;
  _(assert y-x <= 1 && x-y <= 1 && x+y == z)
  return 0;
}
```

2.2 Logical Operators and Quantifiers

VCC provides a number of C extensions that can be used within VCC annotations (such as assertions):

- The Boolean operator ==> denotes logical implication; formally, P ==> Q means ((!P)|| (Q)), and is usually read as "P implies Q". Because ==> has lower precedence than the C operators, it is typically not necessary to parenthesize P or Q.
- The expression \forall T v; E evaluates to 1 if the expression E evaluates to a nonzero value for every value v of type T. For example, the assertion

  ```
  _(assert x > 1 &&
    \forall int i; 1 < i && i < x ==> x % i != 0)
  ```

  checks that (int) x is a prime number. If b is an int array of size N, then

  ```
  _(assert \forall int i; \forall int j;
    0 <= i && i <= j && j < N ==> b[i] <= b[j])
  ```

  checks that b is sorted.
- Similarly, the expression \exists T v; E evaluates to 1 if there is some value of v of type T for which E evaluates to a nonzero value. For example, if b is an int array of size N, the assertion

  ```
  _(assert \exists int i; 0 <= i && i < N && b[i] == 0)
  ```

  asserts that b contains a zero element. \forall and \exists are jointly referred to as ***quantifiers***.
- The type \integer represents the type of mathematical integers; the type \natural denotes the type of natural numbers.
- VCC provides map types; they are declared and used like arrays, but with a type instead of a size. For example, \integer x[int] declares x to be a map from C ints to \integers, and x[-5] is the \integer to which x maps -5. We'll see map types used in § 6.
- Expressions within VCC annotations are restricted in their use of functions: you can only use functions that are proved to be ***pure***, i.e., free from side effects (§ 3.4).

Exercises

1. Write an assertion that says that the int x is a perfect square (i.e., a number being a square of an integer).
2. Write an assertion that says that the int x occurs in the int array b[10].
3. Write an assertion that says that the int array b, of length N, contains no duplicates.
4. Write an assertion that says that all pairs of adjacent elements of the int array b of length N differ by at most 1.
5. Write an assertion that says that an array b of length N contains only perfect squares.

Solutions

1. _(assert \exists unsigned y; x == y*y)
2. _(assert \exists unsigned i; i < 10 && b[i] == x)
3. _(assert \forall unsigned i,j; i < j && j < N ==> b[i] != b[j])
4.
```
_(assert \forall unsigned i,j; 0 <= i && i+1 < N
          ==> -1 <= b[i] - b[i+1] && b[i] - b[i+1] <= 1)
```
5. _(assert \forall unsigned i; i < N ==> \exists unsigned j; b[i] == j * j)

2.3 Assumptions

You can add to what VCC knows at a particular point with a second type of annotation, called an ***assumption***. The assumption _(assume E) tells VCC to ignore the rest of an execution if E fails to hold (i.e., if E evaluates to 0). Reasoning-wise, the assumption simply adds E to what VCC knows for subsequent reasoning. For example:

```
int x, y;
_(assume x != 0)
y = 100 / x;
```

Without the assumption, VCC would complain about possible division by zero. (VCC checks for division by zero because it would cause the program to crash.) Assuming the assumption, this error cannot happen. Since assumptions (like all annotations) are not seen by the compiler, assumption failure won't cause the program to stop, and subsequent assertions might be violated. To put it another way, if VCC verifies a program, it guarantees that in any prefix of an execution where all (user-provided) assumptions hold, all assertions will also hold. Thus, your verification goal should be to eliminate as many assumptions as possible (preferably all of them).

Although assumptions are generally to be avoided, they are nevertheless sometimes useful: (i) In an incomplete verification, assumptions can be used to mark the knowledge that VCC is missing, and to coordinate further verification work (possibly performed by other people). If you follow a discipline of keeping your code in a state where the whole program verifies, the verification state can be judged by browsing the code (without having to run the verifier).

(ii) When debugging a failed verification, you can use assumptions to narrow down the failed verification to a more specific failure scenario, perhaps even to a complete counterexample.

(iii) Sometimes you want to assume something even though VCC can verify it, just to stop VCC from spending time proving it. For example, assuming \false allows VCC to easily prove subsequent assertions, thereby focussing its attention on other parts of the code. Temporarily adding assumptions is a common tactic when developing annotations for complex functions.

(iv) Sometimes you want to make assumptions about the operating environment of a program. For example, you might want to assume that a 64-bit counter doesn't overflow, but don't want to justify it formally because it depends on extra-logical as-

sumptions (like the speed of the hardware or the lifetime of the software).

(v) Assumptions provide a useful tool in explaining how VCC reasons about your program. We'll see examples of this throughout this tutorial.

An assertion can also change what VCC knows after the assertion, if the assertion fails to verify: although VCC will report the failure as an error, it will assume the asserted fact holds afterward. For example, in the following VCC will report an error for the first assumption, but not for the second:

```
int x;
_(assert x == 1)
_(assert x > 0)
```

Exercises

1. In the following program fragment, which assertions will fail?

   ```
   int x,y;
   _(assert x > 5)
   _(assert x > 3)
   _(assert x < 2)
   _(assert y < 3)
   ```

2. Is there any difference between

   ```
   _(assume p)
   _(assume q)
   ```

   and

   ```
   _(assume q)
   _(assume p)
   ```

   ? What if the assumptions are replaced by assertions?

3. Is there any difference between

   ```
   _(assume p)
   _(assert q)
   ```

   and

   ```
   _(assert (!p) || (q))
   ```

   ?

Solutions

1. ```
   int x,y;
   _(assert x > 5) // fails
   _(assert x > 3) // succeeds
   _(assert x < 2) // fails
   _(assert y < 3) // fails
   ```

2. No difference for assumptions. For assertions, the order does matter, because the first assertion might allow the second to succeed.

3. If one of the assertions succeeds, the other will also. However, assuming and then asserting leaves the assumption in effect for the code that follows, whereas asserting the conditional does not.

2.4 Overflows and unchecked arithmetic

Note: this section can be skipped on first reading.

Consider the C expression a+b, when a and b are, say, unsigned ints. This might represent one of two programmer intentions. Most of the time, it is intended to mean ordinary arithmetic addition on numbers; program correctness is then likely to depend on this addition not causing an overflow. However, sometimes the program is designed to cope with overflow, so the programmer means (a + b)% UINT_MAX+1. It is always sound to use this second interpretation, but VCC nevertheless assumes the first by default, for several reasons:

- The first interpretation is much more common.
- The second interpretation introduces an implicit % operator, turning linear arithmetic into nonlinear arithmetic and making subsequent reasoning much more difficult.
- If the first interpretation is intended but the addition can in fact overflow, this potential error will only manifest later in the code, making the source of the error harder to track down.

Here is an example where the second interpretation is intended, but VCC complains because it assumes the first:

```
#include <vcc.h>

unsigned hash(unsigned char *s, unsigned len)
  _(requires \thread_local_array(s, len))
{
  unsigned i, res;
  for (res = 0, i = 0; i < len; ++i)
    res = (res + s[i]) * 13;
  return res;
}
```

```
Verification of hash failed.
testcase(9,11) : error VC8004: (res + s[i]) * 13 might
     overflow.
```

VCC complains that the hash-computing operation might overflow. To indicate that this possible overflow behavior is desired we use _(unchecked), with syntax similar to a regular C type-cast. This annotation applies to the following expression, and indicates that you expect that there might be overflows in there. Thus, replacing the body of the loop with the following makes the program verify:

```
res = _(unchecked)((res + s[i]) * 13);
```

Note that "unchecked" does not mean "unsafe". The C standard mandates the second interpretation for unsigned overflows, and signed overflows are usually implementation-defined to use two-complement. It just means that VCC will loose information about the operation. For example consider:

```
int a, b;
// ...
a = b + 1;
_(assert a < b)
```

This will either complain about possible overflow of b + 1 or succeed. However, the following might complain about a < b, if VCC does not know that b + 1 doesn't overflow.

```
int a, b;
// ...
a = _(unchecked)(b + 1);
_(assert a < b)
```

Think of _(unchecked)E as computing the expression using mathematical integers, which never overflow, and then casting the result to the desired range. VCC knows that

- if the value of E lies within the range of its type, then E == _(unchecked)E;
- if x and y are unsigned, then _(unchecked)(x+y)== (x+y <= UINT_MAX ? x+y : x + y − UINT_MAX), and similarly for other types;
- if x and y are unsigned, then _(unchecked)(x−y)== (x >= y ? x − y : UINT_MAX − y + x + 1), and similarly for other types.
```

If these aren't enough, you will need to resort to bit-vector reasoning (below), or use ghost code to prove what you need [7].

2.4.1 Bitvector Reasoning

Every now and then, you need to prove some low-level fact that VCC can't prove using ordinary logical reasoning. If the fact involves can be expressed over a relatively small number of bits, you can ask VCC to prove it using boolean reasoning at the level of bits, by putting {:bv} after the assert tag. For example:

```
#include <vcc.h>

int min(int a, int b)
  _(requires \true)
  _(ensures \result <= a && \result <= b)
{
  _(assert {:bv} \forall int x; (x & (−1)) == x)
  _(assert {:bv} \forall int a,b; (a − (a − b)) == b)
  return _(unchecked)(a − ((a − b) & −(a > b)));
}
```

Assertions proved in this way cannot mention program variables, and can use only variables of primitive C types.

3. Function Contracts

Next we turn to the specification of functions. We'll take the example from the previous section, and pull the computation of the minimum of two numbers out into a separate function:

```
#include <vcc.h>

int min(int a, int b)
{
  if (a <= b)
    return a;
  else return b;
}

int main()
{
  int x, y, z;
  z = min(x, y);
  _(assert z <= x)
  return 0;
}
```

```
Verification of min succeeded.
Verification of main failed.
testcase(15,12) : error VC9500: Assertion 'z <= x' did
     not verify.
```

(The listing above presents both the source code and the output of VCC, typeset in a different fonts, and the actual file name of the example is replaced with `testcase`.) VCC failed to prove our assertion, even though it's easy to see that it always holds. This is because verification in VCC is ***modular***: VCC doesn't look inside the body of a function (such as the definition of min()) when understanding the effect of a call to the function (such as the call from main()); all it knows about the effect of calling min() is that the call satisfies the specification of min(). Since the correctness of main() clearly depends on what min() does, we need to specify min() in order to verify main().

The specification of a function is sometimes called its ***contract***, because it gives obligations on both the function and its callers. It is provided by four types of annotations:

- A requirement on the caller (sometimes called a ***precondition*** of the function) takes the form _(requires E), where E is an expression. It says that callers of the function promise that E will hold on function entry.
- A requirement on the function (sometimes called a ***postcondition*** of the function) takes the form _(ensures E), where E is an expression. It says that the function promises that E holds just before control exits the function.
- The third type of contract annotation, a ***writes clause***, is described in the next section. In this example, the lack of writes clauses says that min() has no side effects that are visible to its caller.
- The last type of contract annotation, a ***termination clause***, is described in section § 3.3. For now, we won't bother proving that our functions terminate.

For example, we can provide a suitable specification for min() as follows:

```
#include <vcc.h>

int min(int a, int b)
  _(requires \true)
  _(ensures \result <= a && \result <= b)
{
  if (a <= b)
    return a;
  else return b;
}
// ... definition of main() unchanged ...
```

```
Verification of min succeeded.
Verification of main succeeded.
```

(Note that the specification of the function comes after the header and before the function body; you can also put specifications on function declarations (e.g., in header files).) The precondition _(requires \true) of min() doesn't really say anything (since \true holds in every state), but is included to emphasize that the function can be called from any state and with arbitrary parameter values. The postcondition states that the value returned from min() is no bigger than either of the inputs. Note that \true and \result are spelled with a backslash to avoid name clashes with C identifiers.[8]

VCC uses function specifications as follows. When verifying the body of a function, VCC implicitly assumes each precondition of the function on function entry, and implicitly asserts each postcondition of the function (with \result bound to the return value and each parameter bound to its value on function entry) just before the function returns. For every call to the function, VCC replaces the call with an assertion of the preconditions of the function, sets the return value to an arbitrary value, and finally assumes each postcondition of the function. For example, VCC translates the program above roughly as follows:

```
#include <vcc.h>

int min(int a, int b)
{
  int \result;
  // assume precondition of min(a,b)
  _(assume \true)
  if (a <= b)
    \result = a;
```

[7] In particular, you can use iteration or recursion to prove things by induction.

[8] All VCC keywords start with a backslash; this contrasts with annotation tags (like requires), which are only used at the beginning of annotation and therefore cannot be confused with C identifiers (and thus you are still free to have, e.g., a function called requires or assert).

```
  else \result = b;
  // assert postcondition of min(a,b)
  _(assert \result <= a && \result <= b)
}

int main()
{
  int \result;
  // assume precondition of main()
  _(assume \true)
  int x, y, z;
  // z = min(x,y);
  {
    int _res; // placeholder for the result of min()
    // assert precondition of min(x,y)
    _(assert \true)
    // assume postcondition of min(x,y)
    _(assume _res <= x && _res <= y)
    z = _res; // store the result of min()
  }
  _(assert z <= x)
  \result = 0;
  // assert postcondition of main()
  _(assert \true)
}
```

Note that the assumptions above are "harmless", that is in a fully verified program they will be never violated, as each follows from the assertion that proceeds it in an execution[9]For example, the assumption generated by a precondition could fail only if the assertion generated from that same precondition before it fails.

Why modular verification?
Modular verification brings several benefits. First, it allows verification to more easily scale to large programs. Second, by providing a precise interface between caller and callee, it allows you to modify the implementation of a function like min() without having to worry about breaking the verifications of functions that call it (as long as you don't change the specification of min()). This is especially important because these callers normally aren't in scope, and the person maintaining min() might not even know about them (e.g., if min() is in a library). Third, you can verify a function like main() even if the implementation of min() is unavailable (e.g., if it hasn't been written yet).

Exercises

1. What is the effect of giving a function the specification _(requires \false)? How does it effect verification of the function itself? What about its callers? Can you think of a good reason to use such a specification?
2. Can you see anything wrong with the above specification of min()? Can you give a simpler implementation than the one presented? Is this specification strong enough to be useful? If not, how might it be strengthened to make it more useful?
3. Specify a function that returns the (int) square root of its (int) argument. (You can try writing an implementation for the function, but won't be able to verify it until you've learned about loop invariants.)
4. Can you think of useful functions in a real program that might legitimately guarantee only the trivial postcondition _(ensures \true)?

[9] A more detailed explanation of why this translation is sound is given in section § **??**.

Solutions

1. Any function with a spec that includes _(requires \false) should verify. However, a call to such a function will only verify if the call itself is dead code (and VCC can prove it is dead code). Putting _(requires \false) on a function is one way to document that nothing has been proven about it, and that it should not be called.
2. The spec guarantees that min returns a result that is small enough, but nothing prevents it from always returning INT_MIN. This might be strong enough for some applications, but for most, you probably want the additional postcondition _(ensures \result == a || \result == b).
3. ```
   int sqrt(int x)
     _(requires x >= 0)
     _(ensures \result * \result <= x && (result + 1) * (\result + 1)
           > x)
     ;
   ```
4. Many important system functions have no nontrivial guaranteed postcondition, save those that come from the omission of writes clauses. For example, a system call that tries to collect garbage might very well have an empty specification.

3.1 Reading and Writing Memory

When programming in C, it is important to distinguish two kinds of memory access. ***Sequential*** access, which is the default, is appropriate when interference from other threads (or the outside world) is not an issue, e.g., when accessing unshared memory. Sequential accesses can be safely optimized by the compiler by breaking it up into multiple operations, caching reads, reordering operations, and so on. ***Atomic*** access is required when the access might race with other threads, i.e., write to memory that is concurrently read or written, or a read to memory that is concurrently written. Atomic access is typically indicated in C by accessing memory through a volatile type (or through atomic compiler intrinsics). We consider only sequential access for now; we consider atomic access in section § 8.

To access a memory object pointed to by a pointer p, p must point to a valid chunk of memory[10]. (For example, on typical hardware, its virtual address must be suitably aligned, must be mapped to existing physical memory, and so on.) In addition, to safely access memory sequentially, the memory must not be concurrently written by other threads (including hardware and devices). Most of the time[11], this is because the memory object is "part of" something that is "owned" by the thread (concepts that will be discussed later); we express this with the predicate \thread_local(p)[12] VCC asserts this before any sequential memory access to (the memory pointed to by) p.

To write sequentially through p, you need to know \thread_local(p), and that no other thread is trying to read (sequentially or concurrently) through p at the same time[13]. We write this as \mutable(p). Like thread-locality, mutability makes sense only in the context of a particular thread.

[10] VCC actually enforces a stronger condition, that the memory is "typed" according to p.

[11] It is also possible to read memory sequentially if it is a nonvolatile field of an object that is known to be closed, even if it is not owned by the thread; this allows multiple threads to sequentially access shared read-only memory.

[12] Note that thread locality only makes sense in the context of a particular thread, and so cannot appear, for example, in type invariants.

[13] You also need to know that no object invariants depend on *p; this is why object invariants are in effect only for closed objects, and only (parts of) open objects are mutable.
   ```

If VCC doesn't know why an object is thread local, then it has hard time proving that the object stays thread local after an operation with side effects (e.g., a function call). Thus, in preconditions you will sometimes want to use \mutable(p) instead of \thread_local(p). The precise definitions of mutability and thread locality is given in § 5.2, where we also describe another form of guaranteeing thread locality through so called ownership domains.

The NULL pointer, pointers outside bounds of arrays, the memory protected by the operating system, or outside the address space are never thread local (and thus also never mutable nor writable).

There is one further restriction on sequential writes, motivated by the desire to limit the possible side effects of a function call to a specific set of mutable objects. We could do this by adding a postcondition that all other parts of the state are unmodified, but VCC provides some sugar to make specification (and reasoning) about such properties more convenient and efficient. The idea is that when you call a function, you give it permission to write certain objects, but not others; \writable(p) expresses the condition that the function has the right to write to p. Thus, when writing through p, VCC asserts \mutable(p)&& \writable(p).

While mutability is a thread-level concept, writability is a property of a particular instance of an executing function. (That is, just because something is writable for you doesn't mean it will be writable for a function you call.) Therefore, you can't express that a function needs "permission" to write p by _(requires \writable(p)), because preconditions are evaluated in the context of the caller. Instead, you specify that a function needs writability of p at function entry with the annotation _(writes p), called a "writes clause". When you call a function, VCC assumes that all of the objects listed in the writes clauses are writable on function entry. Moreover, if an object becomes mutable (for a thread) after entry to a function call, it is considered writable within that call (as long as it remains mutable).

Let's have a look at an example:

```
void write(int *p)
  _(writes p)
{ *p = 42; }

void write_wrong(int *p)
  _(requires \mutable(p))
{ *p = 42; }

int read1(int *p)
  _(requires \thread_local(p))
{ return *p; }

int read2(int *p)
  _(requires \mutable(p))
{ return *p; }

int read_wrong(int *p)
{ return *p; }

void test_them()
{
  int a = 3, b = 7;
  read1(&a);
  _(assert a == 3 && b == 7) // OK
  read2(&a);
  _(assert a == 3 && b == 7) // OK
  write(&a);
  _(assert b == 7) // OK
  _(assert a == 3) // error
}
```

```
Verification of write succeeded.
Verification of write_wrong failed.
testcase(10,4) : error VC8507: Assertion 'p is
    writable' did not verify.
Verification of read1 succeeded.
Verification of read2 succeeded.
Verification of read_wrong failed.
testcase(21,11) : error VC8512: Assertion 'p is thread
    local' did not verify.
Verification of test_them failed.
testcase(32,12) : error VC9500: Assertion 'a == 3' did
    not verify.
```

The function write_wrong fails because p is only mutable, and not writable. In read_wrong VCC complains that it does not know anything about p (maybe it's NULL, who knows), in particular it doesn't know it's thread-local. read2 is fine because \mutable is stronger than \thread_local. Finally, in test_them the first three assertions succeed because if something is not listed in the writes clause of the called function it cannot change. The last assertion fails, because write() listed &a in its writes clause.

Intuitively, the clause _(writes p, q) says that, of the memory objects that are thread-local to the caller before the call, the function is going to modify only the object pointed to by p and the object pointed to by q. In other words, it is roughly equivalent to a postcondition that ensures that all other objects thread-local to the caller prior to the call remain unchanged. A function can have multiple writes clauses, and implicitly combines them into a single set. If a function spec contains no writes clauses, it is equivalent to specifying a writes clause with empty set of pointers.

Here is a simple example of a function that visibly reads and writes memory; it simply copies data from one location to another.

```
#include <vcc.h>

void copy(int *from, int *to)
  _(requires \thread_local(from))
  _(writes to)
  _(ensures *to == \old(*from))
{
  *to = *from;
}

int z;

void main()
  _(writes &z)
{
  int x,y;
  copy(&x,&y);
  copy(&y,&z);
  _(assert x==y && y==z)
}
```

```
Verification of copy succeeded.
Verification of main succeeded.
```

In the postcondition the expression \old(E) returns the value the expression E had on function entry. Thus, our postcondition states that the new value of *to equals the value *from had on call entry. VCC translates the function call copy(&x,&y) approximately as follows:

```
_(assert \thread_local(&x))
_(assert \mutable(&y))
// record the value of x
int _old_x = x;
// havoc the written variables
```

```
havoc(y);
// assume the postcondition
_(assume y == _old_x)
```

3.1.1 Local Variables

Unlike most block-structured languages, C allows you to take the addresses of local variables (with the & operator). If you take the address of a local, nothing prevents you from storing that address in a data structure, and trying to dereference the address after the lifetime of the variable has ended. (The result is not pretty.) Even if you are careful about its lifetime, once you take the address of a variable, you have to worry that writing through some seemingly unrelated pointer might change the value of the variable, which is a pain.

Because of this, VCC distinguishes between local variables whose addresses are never taken and those whose addresses are taken; the former are called ***purely local*** variables. A purely local variable is much, much easier to reason about; you know that its value can be changed only by a an update through its name (In particular, it cannot be modified by function calls, by assignments to other variables, or assignments through pointers.) Purely local variables are always considered thread-local (so there is no thread-locality check when reading them) and writable (so you never have to mention them in writes clauses of loops or blocks). Also, if you have a loop that doesn't modify a purely local variable in scope, VCC will automatically infer that the value of that variable is not changed in the loop. So you should definitely keep variables purely local whenever possible.

A local variable that is not purely local is treated as if it was allocated on the heap when its lifetime starts (but without the possibility of allocation failure), and is freed when the lifetime ends.

The treatment of pure locality sometimes results in the strange phenomenon that changing some code near the end of a function body can cause verification of someting earlier in the function body to fail. This is because if you take the address of a local near the bottom, it is treated as impure for the whole function body. The simplest workaround in such cases is just to introduce a new local for the bottom part of the function[14].

3.2 Arrays

Array accesses are a kind of pointer accesses. Thus, before allowing you to read an element of an array VCC checks if it's thread-local. Usually you want to specify that all elements of an array are thread-local, which is done using the expression \thread_local_array(ar, sz). It is essentially a syntactic sugar for \forall unsigned i; i < sz ==> \thread_local(&ar[i]). The annotation \mutable_array() is analogous. To specify that an array is writable use:

```
_(writes \array_range(ar, sz))
```

which is roughly a syntactic sugar for:

```
_(writes &ar[0], &ar[1], ..., &ar[sz−1])
```

For example, the function below is recursive implementation of the C standard library memcpy() function:

```
void my_memcpy(unsigned char *dst,
               unsigned char *src,
               unsigned len)
  _(writes \array_range(dst, len))
  _(requires \thread_local_array(src, len))
  _(requires \arrays_disjoint(src, len, dst, len))
  _(ensures \forall unsigned i; i < len
      ==> dst[i] == \old(src[i]))
{
  if (len > 0) {
    dst[len − 1] = src[len − 1];
    my_memcpy(dst, src, len − 1);
  }
}
```

It requires that array src is thread-local, dst is writable, and they do not overlap. It ensures that, at all indices, dst has the value src. The next section presents a more conventional implementation using a loop.

3.3 Termination

A function terminates if it is guaranteed to return to its caller. You can specify termination for simple functions (like the ones we've seen so far) by simply adding to the specification _(decreases 0); this will do the job as long as your functions are not recursive. For functions that are recursive (or which look potentially recursive to VCC, because of potential callbacks from functions whose bodies are hidden from VCC), the termination clause of a function gives a measure that decreases for each call that might start a chain of calls back the function. For example, to verify the termination of my_memcpy above, you need only add to its specification the additional annotation _(decreases len). This annotation defines a "measure" on calls to my_memcpy (namely, the value passed as the last parameter). VCC checks termination by checking that (1) all loops in the body terminate (§ 4.1), and (2) for every function call within the body of my_memcpy that is potentially the first of a chain of calls leading to a call back to my_memcpy, the called function has a _(decreases) specification and the measure of the call to that function is strictly less than the measure of the calling function instance.

It is usually a good idea to prove termination for sequential code when you can[15]. More details on termination measures can be found in the VCC manual.

3.4 Pure functions

A ***pure function*** is one that has no side effects on the program state. In VCC, pure functions are not allowed to allocate memory, and can write only to local variables. Only pure functions can be called within VCC annotations; such functions are required to terminate[16].

The function min() above example of a function that can be declared to be pure; this is done by adding the modifier _(pure) to the beginning of the function specification, e.g.,

```
_(pure) min(int x, int y) ...
```

Being pure is a stronger condition that simply having an empty writes clause. This is because a writes clause has only to mention those side effects that might cause the caller to lose information (i.e., knowledge) about the state, and as we have seen, VCC takes advantage of the kind of information callers maintain to limit the kinds of side effects that have to be reported.

A pure function that you don't want to be executed can be defined using the _(def) tag, which is essentially a pure ghost

[14] This won't effect the final binary produced by a decent optimizing compiler.

[15] You should also consider doing it for your concurrent code, but here VCC is much more limited in its capabilities. The reason for this is that proving termination for a racy function (e.g., one that has to compete for locks) typically depends on fairness assumptions (e.g., that a function trying to grab a spinlock will eventually get lucky and get it, if the spinlock is infinitely often free) and/or global termination measures (e.g., to make sure that other threads will release spinlocks once they acquire them). VCC does not currently support either of these.

[16] This is to guarantee that there is indeed a mathematical function satisfying the specification of the function.

function (one that can be used only in specifications) that is inlined, and uses the following streamlined syntax:

```
_(def \bool sorted(int *arr, unsigned len) {
  return \forall unsigned i, j;
    i <= j && j < len ==> arr[i] <= arr[j];
})
```

A partial spec for a sorting routine could look like the following:[17]

```
void sort(int *arr, unsigned len)
  _(writes \array_range(arr, len))
  _(ensures sorted(arr, len))
```

3.5 Contracts on Blocks

Sometimes, a large function will contain an inner block that implements some simple functionality, but you don't want to refactor it into a separate function (e.g., because you don't want to bother with having to pass in a bunch of parameters, or because you want to verify code without rewriting it). VCC lets you conduct your verification as if you had done so, by putting a function-like specification on the block. This is done by simply writing function specifications preceding the block, e.g.,

```
...
x = 5;
_(requires x == 5)
_(writes &x)
_(ensures x == 6)
{
  x++;
}
...
```

VCC translates this by (internally) refactoring the block into a function, the parameters of which are the variables from the surrounding scope that are mentioned within the block (or the block specifications), so blocks with contracts cannot have statements that transfer control outside of the block. The advantages of this translation is that within the block, VCC can ignore what it knows about the preceding context, and following the block, VCC can "forget" what it knew inside the block (other than what escapes through the ensures clauses); in each case, this results in less distracting irrelevant detail for the theorem prover.

Sometimes, you don't care about information flowing into the block, but only care about the information flowing out of the block. In this case, you can use the precondition _(requires \full_context()), which tells VCC to verify the block using all of the information about what came before, but using the postconditions and writes clauses to hide information about what went on inside the block to the code that follows the block.

4. Loop invariants

For the most part, VCC computes what it knows at a control point from what it knows at earlier control points. This works even if there are gotos from earlier control points; VCC just takes the disjunction of what it knows for each of the possible places it came from. But when the control flow contains a loop, VCC faces a chicken-egg problem, since what it knows at the top of the loop (i.e., at the beginning of each loop iteration) depends not only on what it knew just before the loop, but also on what it knew just before it jumped back to the top of the loop from the loop body.

Rather than trying to guess what it should know at the top of a loop, VCC lets you tell it what it should know, by providing ***loop invariants***. To make sure that loop invariants indeed hold whenever control reaches the top of the loop, VCC asserts that the invariants hold wherever control jumps to the top of the loop – namely, on loop entry and at the end of the loop body.

Let's look at an example:

```
#include <vcc.h>

void divide(unsigned x,
            unsigned d,
            unsigned *q,
            unsigned *r)
_(requires d > 0 && q != r)
_(writes q,r)
_(ensures x == d*(*q) + *r && *r < d)
{
  unsigned lq = 0;
  unsigned lr = x;
  while (lr >= d)
  _(invariant x == d*lq + lr)
  {
    lq++;
    lr -= d;
  }
  *q = lq;
  *r = lr;
}
/*{end}*/
```

```
Verification of divide succeeded.
```

The divide() function computes the quotient and remainder of integer division of x by d using the classic division algorithm. The loop invariant says that we have a suitable answer, except with a remainder that is possibly too big. VCC translates this example roughly as follows:

```
#include <vcc.h>

void divide(unsigned x,
            unsigned d,
            unsigned *q,
            unsigned *r)
  _(writes q,r)
{
  // assume the precondition
  _(assume d > 0 && q != r)
  unsigned lq = 0;
  unsigned lr = x;

  // check that the invariant holds on loop entry
  _(assert x == d*lq + lr)

  // start an arbitrary iteration
  // forget variables modified in the loop
  {
    unsigned _fresh_lq, _fresh_lr;
    lq = _fresh_lq; lr = _fresh_lr;
  }
  // assume that the loop invariant holds
  _(assume x == d*lq + lr)
  // jump out if the loop terminated
  if (!(lr >= d))
    goto loopExit;
  // body of the loop
  {
    lq++;
    lr -= d;
  }
  // check that the loop preserves the invariant
  _(assert x == d*lq + lr)
  // end of the loop
  _(assume \false)
```

[17] We will take care about input being permutation of the output in § 6.

```
  loopExit:
  *q = lq;
  *r = lr;
  // assert postcondition
  _(assert x == d*(*q) + *r && *r < d)
}
```

Note that this translation has removed all cycles from the control flow graph of the function (even though it has gotos); this means that VCC can use the rules of the previous sections to reason about the program. In VCC, all program reasoning is reduced to reasoning about acyclic chunks of code in this way.

Note that the invariant is asserted wherever control moves to the top of the loop (here, on entry to the loop and at the end of the loop body). On loop entry, VCC forgets the value of each variable modified in the loop (in this case just the local variables lr and ld), and assumes the invariant (which places some constraints on these variables). VCC doesn't have to consider the actual jump from the end of the loop iteration back to the top of the loop (since it has already checked the loop invariant), so further consideration of that branch is cut off with _(assume \false). Each loop exit is translated into a goto that jumps to just beyond the loop (to loopExit). At this control point, we know the loop invariant holds and that lr < d, which together imply that we have computed the quotient and remainder.

For another, more typical example of a loop, consider the following function that uses linear search to determine if a value occurs within an array:

```
#include <vcc.h>
#include <limits.h>

unsigned lsearch(int elt, int *ar, unsigned sz)
  _(requires \thread_local_array(ar, sz))
  _(ensures \result != UINT_MAX
      ==> ar[\result] == elt)
  _(ensures \forall unsigned i;
     i < sz && i < \result ==> ar[i] != elt)
{
  unsigned i;
  for (i = 0; i < sz; i++)
    _(invariant \forall unsigned j;
        j < i ==> ar[j] != elt)
  {
    if (ar[i] == elt) return i;
  }
  return UINT_MAX;
}
/*{end}*/
```

```
Verification of lsearch succeeded.
```

The postconditions say that the returned value is the minimal array index at which elt occurs (or UINT_MAX if it does not occur). The loop invariant says that elt does not occur in ar[0]. . . ar[i − 1].

4.1 Termination measures for loops

To prove that a loop terminates, it can be given a _(decreases) clause, just as a function can. Before control returns from inside the loop to the top of the loop, there is an implicit assertion that the measure on the loop has gone down from its value at the beginning of the iteration. (Note that if the loop body contains a function call, its measure is checked against the measure assigned to the function, not to the loop.)

For example, in the divide function, we could specify that the loop terminates by adding the specification _(decreases lr) to the loop specification. This would then allow us to add the specification _(decreases 0) to the divide function itself.

If a function with a termination measure contains a for loop without a termination measure, VCC tries to guess one from syntactic form of the loop header. Thus, most boilerplate for loops do not require explicit termination measures.

Here's an example, a function that sorts an array using bubblesort. VCC infers a termination measure for the outer loop, but currently needs to be given one for the inner loop:

```
_(def \bool sorted(int *buf, unsigned len) {
  return \forall unsigned i, j; i < j && j < len
           ==> buf[i] <= buf[j];
})

void sort(int *buf, unsigned len)
  _(writes \array_range(buf, len))
  _(ensures sorted(buf, len))
  _(decreases 0)
{
  if (len < 2) return;
  for (unsigned i = len; i > 0; i--)
    _(invariant i <= len)
    _(invariant \forall unsigned u,v;
      i <= v && u <= v && v < len
      ==> buf[u] <= buf[v])
    for (unsigned j = 0; j + 1 < i; j++)
      _(decreases i-j)
      _(invariant j < i)
      _(invariant \forall unsigned u,v;
          i <= v && u <= v && v < len
          ==> buf[u] <= buf[v])
      _(invariant \forall unsigned u; u < j
          ==> buf[u] <= buf[j])
      if (buf[j] > buf[j+1]) {
        int tmp = buf[j];
        buf[j] = buf[j+1];
        buf[j+1] = tmp;
      }
}
```

The specification that we use is that the output of the sorting routine is sorted. However, it doesn't say that the output is a permutation of the input. We'll show how to do that in § 6.2.

4.2 Writes clauses for loops

Loops are in many ways similar to recursive functions. Invariants work as the combination of pre- and post-conditions. Similarly to functions loops can also have writes clauses. You can provide a writes clause using exactly the same syntax as for functions. When you do not write any heap location in the loop (which has been the case in all examples so far), VCC will automatically infer an empty writes clause. Otherwise, it will take the writes clause that is specified on the function. So by default, the loop is allowed to write everything that the function can. Here is an example of such implicit writes clause, a reinterpretation of my_memcpy() from § 3.2.

```
void my_memcpy(unsigned char *dst,
               unsigned char *src,
               unsigned len)
  _(writes \array_range(dst, len))
  _(requires \thread_local_array(src, len))
  _(requires \arrays_disjoint(src, len, dst, len))
  _(ensures \forall unsigned i; i < len ==>
      dst[i] == \old(src[i]))
  _(decreases 0)
{
  unsigned k;
  for (k = 0; k < len; ++k)
    _(invariant \forall unsigned i; i < k ==>
        dst[i] == \old(src[i]))
  {
```

```
    dst[k] = src[k];
  }
}
```

(Note that VCC also inferred an appropriate termination measure for the for loop.)

If a loop does not write everything the function can write you will often want to provide explicit write clauses. Here's a variation of memcpy(), which clears (maybe for security reasons) the source buffer after copying it.

```
void memcpyandclr(unsigned char *dst,
                  unsigned char *src,
                  unsigned len)
  _(writes \array_range(src, len))
  _(writes \array_range(dst, len))
  _(requires \arrays_disjoint(src, len, dst, len))
  _(ensures \forall unsigned i; i < len
      ==> dst[i] == \old(src[i]))
  _(ensures \forall unsigned i; i < len
      ==> src[i] == 0)
  _(decreases 0)
{
  unsigned k;
  for (k = 0; k < len; ++k)
    _(writes \array_range(dst, len))
    _(invariant \forall unsigned i; i < k
        ==> dst[i] == \old(src[i]))
  {
    dst[k] = src[k];
  }
  for (k = 0; k < len; ++k)
    _(writes \array_range(src, len))
    _(invariant \forall unsigned i; i < k
        ==> src[i] == 0)
  {
    src[k] = 0;
  }
}
```

If the second loops did not provide a writes clause, we couldn't prove the first postcondition—VCC would think that the second loop could have overwritten dst.

Exercises

Specify and verify iterative implementations of the following functions:

1. a function that takes two arrays and checks whether the arrays are equal (i.e., whether they contain the same sequence of elements);
2. a function that checks whether two sorted arrays contain a common element;
3. a function that checks whether a sorted array contains a given value;
4. a function that takes an array and checks whether it contains any duplicate elements;
5. a function that takes an array and reverses it.

Solutions can be found in the file 5.7.solutions.c in the tutorial directory.

5. Object invariants

Pre- and postconditions allow specification that a function requires or ensures that data is in a consistent state on entry to or exit from the function. However, it is usually better practice to associate such consistency with the data itself. This is particularly important for data accessed concurrently by multiple threads (where data must be kept in a consistent state at all times), but even for sequential programs enforcing consistency conditions on data reduces annotation clutter and allows for introduction of abstraction boundaries.

In VCC, you can associate ***object invariants*** with compound C types (structs and unions). The invariants of a type describe how "good" objects of that type behave. In this section and the next, we consider only the static aspects of this behavior, namely what the "good" states of an object are. Invariants can also constrain how how "good" objects can change, are covered in § 8.

As a first example, consider the following type definition of '\0'-terminated safe strings implemented with statically allocated arrays (we'll see dynamic allocation later).

```
#define SSTR_MAXLEN 100
typedef struct SafeString {
  unsigned len;
  char content[SSTR_MAXLEN + 1];
  _(invariant \this->len <= SSTR_MAXLEN)
  _(invariant content[len] == '\0')
} SafeString;
```

The invariant of SafeString states that a good SafeString has length not more than SSTR_MAXLEN and is '\0'-terminated. Within a type invariant, \this is a pointer to the current instance of the type (as in the first invariant), but fields can also be referred to directly (as in the second invariant).

Because memory in C is allocated without initialization, no non-trivial object invariant is guaranteed to hold on allocation. An object that is known to be in a good state (which implies its invariants hold) is said to be ***closed***, while one that is now known to be in a good state is said to be ***open***. A mutable object is just an open object owned by the current thread; a ***wrapped*** object is a closed object owned by the current thread. A function serving as a constructor for a type will normally establish invariant and wrap the object:

```
void sstr_init(struct SafeString *s)
  _(writes \span(s))
  _(ensures \wrapped(s))
  _(decreases 0)
{
  s->len = 0;
  s->content[0] = '\0';
  _(wrap s)
}
```

For a pointer p of structured type, \span(p) returns the set of pointers to members of p. Arrays of base types produce one pointer for each base type component, so in this example, \span(s) abbreviates the set

```
{ s, &s->len, &s->content[0], &s->content[1], ...,
    &s->content[SSTR_MAXLEN] }
```

Thus, the writes clause says that the function can write the fields of s. The postcondition says that the function returns with s wrapped, which implies also that the invariant of s holds; this invariant is checked when the object is wrapped. (You can see this check fail by commenting out any of the assignment statements.)

A function that modifies a wrapped object must first unwrap it, make the necessary updates, and wrap the object again (which causes another check of the object invariant); this is because VCC does not allow (nonvolatile) fields of an object while it is closed. Unwrapping an object adds all of its members to the writes set of a function, so such a function has to report that it writes the object, but does not have to report writing the fields of the object:

```
void sstr_append_char(struct SafeString *s, char c)
  _(requires \wrapped(s))
  _(requires s->len < SSTR_MAXLEN)
  _(ensures \wrapped(s))
```

```
  _(writes s)
  _(decreases 0)
{
  _(unwrap s)
  s->content[s->len++] = c;
  s->content[s->len] = '\0';
  _(wrap s)
}
```

Finally, a function that only reads an object need not unwrap, and so need not list it in its writes clause. For example:

```
int sstr_index_of(struct SafeString *s, char c)
  _(requires \wrapped(s))
  _(ensures \result >= 0 ==> s->content[\result] == c)
  _(decreases 0)
{
  unsigned i;
  for (i = 0; i < s->len; ++i)
          _(decreases s->len - i)
    if (s->content[i] == c) return (int)i;
  return -1;
}
```

VCC keeps track of whether an object is closed with the \bool field \closed (which is a field of every object). It keeps track of the owner of an object with the field \owner. This field is a pointer to an object, which might be a thread.

```
_(def \bool \wrapped(\object o) {
    return \non_primitive_ptr(o) && o->\closed && o->\owner ==
        \me;
})
_(def \bool \mutable(\object o) {
    if (!\non_primitive_ptr(o)) return \mutable(\embedding(o));
    return !o->\closed && o->\owner == \me;
}
```

These definitions use several new features:

- When verifying a body of a function VCC assumes that it is being executed by some particular thread. The \thread object representing it is referred to as \me.
- The type \object is much like void*, in the sense that it is a wildcard for any pointer type. However, while casting a pointer to void * causes information about its type to be lost, casting to \object does not. Note that (sadly) \object includes not just pointers to objects, but also pointers to primitive types like int that can be fields of objects but are not first-class objects. What we call "objects" are \objects for which the function \non_primitive_ptr returns \true.
- When applied to a pointer that is not an object, \embedding(o) returns the object of which o is a field. Unlike pointers in C, VCC pointers include information about the object of which they are a field.

5.1 Wrap/unwrap protocol

We now consider wrapping and unwrapping in more detail.

If an object is owned by a thread, only that thread can change its (nonvolatile) fields (and then only if the object is open), wrap or unwrap it, or change its owner to another object. Objects are guaranteed to be closed when they are owned by objects other than threads, and only closed objects can own other objects. Finally, threads are always closed, and own themselves.

The call _(unwrap o) translates essentially to the following:

1. _(assert \wrapped(o));
2. _(assert \writable(o)), i.e., that o was either listed in the writes clause of the function, or became \wrapped after the current function activation started;
3. assume the invariant of o
4. set o->\closed = \false;
5. add the span of the object (i.e., all its fields) to the writes set;
6. set \me to be the owner of any objects owned by o;
7. assert that the transition did not violate any invariants of o of the form _(invariant _(on_unwrap(\this,p))).

The operation _(wrap o) does the reverse:

1. assert that o is mutable;
2. assert that all objects whose ownership is to be transfered to o (to be defined later) are \wrapped and \writable;
3. set o->\closed = \true;

VCC also provides the syntax

```
_(unwrapping o) { ... }
```

which is equivalent to:

```
_(unwrap o) { ... } _(wrap o)
```

The assert/assume desugaring of the sstr_append_char() function is roughly as follows:

```
void sstr_append_char(struct SafeString *s, char c)
  _(requires \wrapped(s))
  _(requires s->len < SSTR_MAXLEN)
  _(ensures \wrapped(s))
{
  // _(unwrap s), steps 1-5
  _(assert \writable(s))
  _(assert \wrapped(s))
  _(assume s->len <= SSTR_MAXLEN &&
           s->content[s->len] == '\0')
  _(ghost s->\closed = \false)
  _(assume \writable(\span(s)))

  s->content[s->len++] = c;
  s->content[s->len] = '\0';

  // _(wrap s), steps 1-3
  _(assert \mutable(s))
  _(assert s->len <= SSTR_MAXLEN &&
           s->content[s->len] == '\0')
  _(ghost s->\closed = \true)
}
```

5.2 Ownership trees

Objects often stand for abstractions that are implemented with more than just one physical object. As a simple example, consider our SafeString, changed to have a dynamically allocated buffer. The logical string object consists of the control object holding the length and the array of bytes holding the content:

```
struct SafeString {
  unsigned capacity, len;
  char *content;
  _(invariant len < capacity)
  _(invariant content[len] == '\0')
  _(invariant \mine((char[capacity])content))
};
```

In C the type char[10] denotes an array with exactly 10 elements. VCC extends that location to allow the type char[capacity] denoting an array with capacity elements (where capacity is an expression). Such types can be only used in casts (in annotations). For example, (char[capacity])content means to take the pointer content and interpret it as an ***array object*** consisting of capacity elements of type char.

The invariant of SafeString specifies that it ***owns*** the array object. (The use of an array object is necessary; chars are not first class objects (they can only be fields of other objects) and so cannot have owners.) The syntax \mine(o1, ..., oN) is roughly equivalent to

```
o1−>\owner == \this && ... && oN−>\owner == \this
```

Conceptually there isn't much difference between content being an actual field of the SafeString (as it was in the previous definition) and it being an array object owned by the SafeString. In particular, in neither case does a function operating on a \wrapped SafeString s have to list s−>content (or any other objects owned by s) in their writes clauses. This is because modifying s−>content requires first unwrapping s, and doing so adds s−>content to the writes set. For example:

```
void sstr_append_char(struct SafeString *s, char c)
  _(requires \wrapped(s))
  _(requires s−>len < s−>capacity − 1)
  _(ensures \wrapped(s))
  _(writes s)
  _(decreases 0)
{
  _(unwrapping s) {
    _(unwrapping (char[s−>capacity])(s−>content)) {
      s−>content[s−>len] = c;
      s−>len++;
      s−>content[s−>len] = '\0';
    }
  }
}
```

Let cont = (char[s−>capacity])s−>content. At the beginning of the function, s is owned by the current thread (\me) and closed (i.e., \wrapped), whereas (by the string invariant) cont is owned by s (and therefore closed). Unwrapping s transfers ownership of cont to \me, but cont remains closed. Thus, unwrapping s makes the string mutable, and cont wrapped. Then we unwrap cont (which doesn't own anything, so the thread gets no new wrapped objects), perform the changes, and wrap cont. Finally, we wrap s. This transfers ownership of cont from the current thread to s, so cont is no longer wrapped (but still closed). Here is the assert/assume translation:

```
void sstr_append_char(struct SafeString *s, char c)
  _(requires \wrapped(s))
  _(requires s−>len < s−>capacity − 1)
  _(ensures \wrapped(s))
  _(writes s)
{
  _(ghost \object cont = (char[s−>capacity]) s−>content; )
  // _(unwrap s) steps 1−5
  _(assert \writable(s) && \wrapped(s))
  _(assume \writable(\span(s)) && \inv(s))
  _(ghost s−>\closed = \false; )
  // and the transfer:
  _(ghost cont−>\owner = \me; )
  _(assume \writable(cont))
  // _(unwrap cont) steps 1−5
  _(assert \writable(cont) && \wrapped(cont))
  _(ghost cont−>\closed = \false; )
  _(assume \writable(\span(cont)) && \inv(cont))
  // no transfer here

  s−>content[s−>len++] = c;
  s−>content[s−>len] = '\0';

  // _(wrap cont) steps 1−3
  _(assert \mutable(cont) && \inv(cont))
  _(ghost cont−>\closed = \true; )
  // _(wrap s) steps 1−3, with transfer in the middle
  _(assert \mutable(s))
  _(ghost cont−>\owner = s; )
  _(assert \inv(s))
  _(ghost s−>\closed = \true; )
}
```

Here, \inv(p) means the (user-defined) invariant of object p. There are two ownership transfers of cont to and from \me because s owns cont beforehand, as specified in its invariant. However, suppose we had an invariant like the following:

```
struct S {
  struct T *a, *b;
  _(invariant \mine(a) || \mine(b))
};
```

When wrapping an instance of struct S, VCC wouldn't know which object to transfer ownership of to the instance. Therefore, VCC rejects such invariants, and only allow \mine(...) as a top-level conjunct in an invariant, unless further annotation is given; see § 5.3.

5.3 Dynamic ownership

When a struct is annotated with _(dynamic_owns) the ownership transfers during wrapping need to performed explicitly, but \mine(...) can be freely used in its invariant, including using it under a universal quantifier.

```
_(dynamic_owns) struct SafeContainer {
  struct SafeString **strings;
  unsigned len;

  _(invariant \mine((struct SafeString *[len])strings))
  _(invariant \forall unsigned i; i < len ==>
      \mine(strings[i]))
  _(invariant \forall unsigned i, j; i < len && j < len ==>
      i != j ==> strings[i] != strings[j])
};
```

The invariant of struct SafeContainer states that it owns its underlying array, as well as all elements pointed to from it. It also says that there are no duplicates in that array. Suppose o is a SafeContainer and we want to change o−>strings[idx], from x to y. After such an operation, the container should own whatever it used to own, minus x, plus y. To facilitate this, VCC provides a field \owns for each object with the implicit invariant

```
_(invariant \this−>\closed ==> \forall \object q;
  (q \in \this−>\owns <==> q−>\owner == \this)
```

That is, for closed p, the set p−>\owns contains exactly the objects that have p as their owner. The opertions _(wrap p) and _(unwrap p) do not change p−>\owns, so _(wrap p) attempts to transfer to p ownership of all object in p−>\owns, which (as described previously) causes an assertion that all of these objects are \wrapped and \writable.

Thus, the usual pattern is to unwrap o, potentially modify fields of o and o−>\owns, and wrap o. Note that when no ownership transfers are needed, one can just unwrap and wrap O, without worrying about what o owns. Here is an example with an ownership transfer:

```
void sc_set(struct SafeContainer *c,
            struct SafeString *s, unsigned idx)
  _(requires \wrapped(c) && \wrapped(s))
  _(requires idx < c−>len)
  _(ensures \wrapped(c))
  _(ensures c−>strings[idx] == s)
  _(ensures \wrapped(\old(c−>strings[idx])))
  _(ensures \fresh(\old(c−>strings[idx])))
  _(ensures c−>len == \old(c−>len))
  _(writes c, s)
  _(decreases 0)
{
```

```
  _(assert !(s \in c->\owns))
  _(unwrapping c) {
    _(unwrapping (struct SafeString *[c->len])(c->strings)) {
      c->strings[idx] = s;
    }
    _(ghost {
      c->\owns -= \old(c->strings[idx]);
      c->\owns += s;
    })
  }
}
```

The sc_set() function transfers ownership of s to c, and additionally leaves object initially pointed to by s->strings[idx] wrapped. Moreover, it promises that this object is ***fresh***, i.e., the thread did not own it directly before. This can be used at a call site:

```
void use_case(struct SafeContainer *c, struct SafeString *s)
  _(requires \wrapped(c) && \wrapped(s))
  _(requires c->len > 10)
  _(writes c, s)
{
  struct SafeString *o;
  o = c->strings[5];
  _(assert \wrapped(c)) // OK
  _(assert \wrapped(s)) // OK
  _(assert o \in c->\owns) // OK
  _(assert \wrapped(o)) // error
  sc_set(c, s, 5);
  _(assert o != s) // OK
  _(assert \wrapped(c)) // OK
  _(assert \wrapped(s)) // error
  _(assert \wrapped(o)) // OK
}
```

In the contract of sc_add the string s is mentioned in the writes clause, but in the postcondition we do not say it's wrapped. Thus, asserting \wrapped(s) after the call fails. On the other hand, asserting \wrapped(o) fails before the call, but succeeds afterwards. Additionally, \wrapped(c) holds before and after as expected.

5.4 Ownership domains

The ***sequential ownership domain*** of an object o (written \domain(o)) consists of o along with[18] the union of the ownership domains of all objects owned by o. In other words, it's the set of objects that are transitively owned by o. For most purposes, it is useful to think of o as "containing" all of \domain(o); indeed, if o1 != o2 and neither o1 nor o2 are in the other's \domain, their \domains are necessarily disjoint. In particular, if o1 and o2 are owned by threads then (because threads own themselves) o1 and o2 are necessarily disjoint.

Writability of o gives a thread potential access to all of \domain(o): writability allows the thread to unwrap o, which makes writable both the fields of o and any objects that were owned by o. Conversely, a call to a function that does not list a wrapped object o in its writes clause is guaranteed to leave all of \domain(o) unchanged[19]. However, VCC will only reason about the unchangedness of \domain(o) if it is explicitly brought to its attention, as in the following example:

```
void f(T *p)
  _(writes p) { ... }
...
T *p, *q, *r;
_(assert \wrapped(q) && q != p)
_(assert q \in \domain(q))
_(assert r \in \domain(q))
f(p);
```

The second and third assertions bring to VCC's attention that as long as q is not explicitly unwrapped or included in the writes clause of a function call, r and its fields will not change.

5.5 Simple sequential admissibility

Until now we've ignored the issue of constraints on the form of object invariants. When VCC checks an atomic update to the state, it checks only the invariants of those objects that are actually updated. Constraints on the invariants of other objects are needed to guarantee that invariants of closed unupdated objects are also preserved.

Fortunately, the most common case is trivial: if an invariant of o mentions only (nonvolatile) fields of objects in \domain(o), the invariant is necessarily admissible. More sophisticated kinds of invariants are discussed in § 8.3.

5.6 Type safety

In modern languages like Java, C#, and ML, where memory consists of a collection of typed objects. Programs in these languages allocate objects (as opposed to memory), and the type of an object remains fixed until the object is destroyed. Moreover, a non-null reference to an object is guaranteed to point to a "valid" object. But in C, a type simply provides a way to interpret a sequence of bytes; nothing prevents a program from having multiple pointers of different types pointing into the same physical memory, or having a non-null pointer point into an invalid region of memory.

That said, most C programs really do access memory using a strict type discipline and tacitly assume that their callers do so also. For example, if the parameters of a function are a pointer to an int and a pointer to a char, we shouldn't have to worry about crazy possibilities like the char aliasing with the second half of the int. (Without such assumptions, we would have to provide explicit preconditions to this effect.) On the other hand, if the second parameter is a pointer to an int, we do consider the possibility of aliasing (as we would in a strongly typed language). Moreover, since in C objects of structured types literally contain fields of other types, if the second argument were a struct that had a member of type int, we would have to consider the possibility of the first parameter aliasing that member.

To support this kind of antialiasing by default, VCC essentially maintains a typed view of memory, as follows. Each object has a \bool field \valid. Each object requires a (possibly empty) set of real memory addresses for its representation; this set of addresses constitutes the "footprint" of an objecct. VCC maintains the invariant that in any state, the footprints of distinct valid objects do not overlap. It maintains this in a very simple way: it only makes an object valid by simultaneously making invalid a set of objects, the union of whose footprints include the footprint of the newly valid object. This invariant allows us to reason completely in terms of objects, without every having to worry about aliasing between one object and another[20]. Only valid fields (and purely local variables) can be accessed by a program.

There are rare situations where a program needs to change the type of memory, i.e., make one object invalid while making valid an object that aliases with it. The most common example is in the memory allocator, which needs to create and destroy objects of arbitrary types from arrays of bytes in its memory pool. Therefore, VCC includes annotations (explained in § 7) that explicitly change

[18] The domains of the objects owned by o are included only if o is not declared as _(volatile_owns); see § 8.

[19] This applies to nonvolatile fields of objects in the domain; volatile fields might change silently (see section § 8).

[20] This does not completely remove the need to occasionally reason about address arithmetic, since VCC sometimes has to reason about the relationships between the address of, say, and object o and an object o–f nested within o.

object validity (and are in fact the only means to do so). Thus, while your program can access memory using pretty much arbitrary types and typecasting, doing so is likely to require additional annotations. But for most programs, checking type safety is completely transparent, so you don't have to worry about it.

6. Ghosts

VCC methodology makes heavy use of ***ghost*** data and code - data and code that are not seen by the C compiler (and therefore are not part of the executable), but are included to aid reasoning about the program. Part of the VCC philosophy is that programmers would rather writing extra code than have to drive interactive theorem provers, so ghosts are the preferred way to explain to VCC why your program works.

We have already seen some ghost fields of ordinary data structures (e.g. \closed, \owner, \ownsm \valid, etc.) as well as built-in pieces of ghost code (e.g. _(wrap ...) and _(unwrap ...)). This section is about how to write your own ghost code.

Our first tyical use of ghost data is to express data abstaction. If you implement an abstract set as a list, it is good practice to expose to the client only the set abstraction, while keeping the list implementation private to the implementation of the data type. One way to do this is to store the abstract value in a ghost field and update it explicitly in ghost code when operating on the data structure[21]. Functions operating on the data structure are specified in terms of their effect on the abstract value only; the connection between the abstract and concrete values is written as an invariant of the data structure.

VCC's mathematical types are usually better suited to representing these abstract values than C's built-in types. Here, we will use VCC maps. Recal that a declaration int m[T*] defines a map m from T* to int; for any pointer p of type T* m[p] gives the int to which m maps p. A map bool s[int] can be thought of as a set of ints: the operation s[k] will return true if and only if the element k is in the set s.

For example, here is a simple example of a set of ints implemented with an array:

```
#define SIZE 1000
typedef struct ArraySet {
  _(ghost \bool mem[int]) // abstract value
```

The map mem gives the abstract value of the set. The map idx says where to find each abstract member of the set. We could have eliminated idx and instead used existential quantification to say that every member of the abstract set occurs somewhere in the list. The disadvantage of using an explicit witness like idx is that we have to update it appropriately. The advantage is that the prover doesn't have to guess how to instantiate such existential quantifiers, which makes the verification more robust.

Here is the initializer for these sets:

```
void arraySetInit(ArraySet *s)
  _(requires \extent_mutable(s))
  _(writes \extent(s))
  _(ensures \wrapped(s) && s->mem == \lambda int i; \false)
{
  s->len = 0;
  _(ghost s->mem = \lambda int i; \false)
  _(wrap s)
}
```

[21] An alternative approach is to write a pure ghost function that takes a concrete data structure and returns its abstract value. The disadvantage of this approach is that for recursive structures like lists, the abstraction function is likewise recursive, and so reasoning with it requires substantial guidance from the user.

The standard form of a constructor is to take some raw memory[22], and wrap it, while establishing some condition on its abstract value. Values of maps are constructed using ***lambda expressions***. The expression \lambda T x; E returns a map, which for any x returns the value of expression E (which can mention x). If S is the type of E, then this map is of type S[T].

Here is the membership test:

```
_(pure) BOOL arraySetMem(ArraySet *s, int v)
  _(reads &s->mem)
  _(requires \wrapped(s))
  _(ensures \result == s->mem[v])
{
  for (unsigned i = 0; i < s->len; i++)
    _(invariant \forall unsigned j; j < i
        ==> s->data[j] != v)
  {
    if (s->data[i] == v) return TRUE;
  }
  return FALSE;
}
```

As usual, an accessor is marked as _(pure), and reads only the abstract value of the set. Finally, here is a function that adds a new element to the set:

```
BOOL ArraySetAdd(ArraySet *s, int v)
  _(requires \wrapped(s))
  _(writes s)
  _(ensures \result ==> s->mem ==
      \lambda int i; \old(s->mem[i]) || i == v)
  _(ensures !\result ==> \unchanged(s->mem))
{
  if (s->len == SIZE) return FALSE;
  _(unwrapping s) {
    s->data[s->len] = v;
    _(ghost s->mem[v] = \true)
    _(ghost s->idx[v] = s->len)
    s->len++;
  }
  return TRUE;
}
```

Note that in addition to updating the concrete representation, we also update the abstract value and the witness. This example shows another way to update a map, using array syntax; if m is a variable of type S[T], e is of type T, and e2 is of type S, then the statement m[e1] = e2 abbreviates the statement m = \lambda T v; v == e1 ? e2 : m[v].

Exercises

1. Extend ArraySet with a function that deletes a value from the set.
2. Modify ArraySet to keep the set without duplication.
3. Modify ArraySet to keep the elements ordered. Use binary search to check for membership and for insertion of a new element.
4. Extend ArraySet with a function that adds the contents of one set into another. (Try to calculate the size of the combined list before modifying the target, so that you can fail gracefully.)

6.1 Linked Data Structures

As an example of a more typical dynamic data structure, consider the following alternative implementation of int sets as lists:

```
typedef struct Node {
```

[22] In C, it is normal for constructors to take a pointer to raw memory, to that they can be used to make objects that are embedded within data structures.

```
  struct Node *nxt;
  int data;
} Node;

typedef _(dynamic_owns) struct List {
  Node *head;
  _(ghost \bool val[int];)
  _(ghost Node *find[int];)
  _(invariant head != NULL ==> \mine(head))
  _(invariant \forall Node *n; \mine(n) && n->nxt
      ==> \mine(n->nxt))
  _(invariant \forall Node *n; \mine(n)
      ==> val[n->data])
  _(invariant \forall int v; val[v]
      ==> \mine(find[v]) && find[v]->data == v)
} List;
```

The invariant states that:

- the list owns the head node (if it's non-null)
- if the list owns a node, it also owns the next node (provided it's non-null)
- if the list owns a node n then n->data is in val;
- if v is in val, then it is the data for some node (find[v]) owned by the list.

Note that we have chosen to put all of the list invariants in the List data structure, rather than in the nodes themselves (which would also work). A disadvantage of putting all of the invariants in the List type is that when you modify one of the nodes, you have to check these invariants for all of the nodes (although the invariants are easy to discharge for nodes that are not immediate neighbors). Some advantages of this choice is that it is easier to modify all of the nodes as a group, and that the same Node type can be used for data structures with different invariants (e.g., cyclic lists).

Here is the implementation of the add function:

```
{
  Node *n = malloc(sizeof(*n));
  if (n == NULL) return -1;
  _(unwrapping l) {
    n->nxt = l->head;
    n->data = k;
    _(wrap n)
    l->head = n;
    _(ghost {
      l->\owns += n;
          l->val = (\lambda int z; z == k || l->val[z]);
          l->find = (\lambda int z; z == k ? n : l->find[z]);
    })
  }
  return 0;
}
/*{out}*/
```

```
Verification of List#adm succeeded.
Verification of mklist succeeded.
Verification of add succeeded.
```

We allocate the node, unwrap the list, initialize the new node and wrap it, and prepend the node at the beginning of the list. Then we update the owns set to include the new node, update the abstract value val and the witness find, and finally wrap the list up again (when exiting the _(unwrapping) block).

The invariants of our list say that the abstract value contains exactly the data fields of nodes owned by the list. It also said that pointers from list nodes take you to list nodes. But it doesn't say that every node of the list can be reached from the first node; the invariants would hold if, in addition to those nodes, the list also owned some unrelated cycle of nodes. As a result, the natural algorithm for checking if a value is in the set (by walking down the list) won't verify; if it finds a value, its data is guaranteed to be in the set, but not vice-versa. Moreover, the invariants aren't strong enough to guarantee that the list itself is acyclic.

Thus, we need to strengthen the invariant of the list to guarantee that every node of the list is reachable from the head. This cannot be expressed directly with first-order logic, but there are several ways to express this using VCC using ghost data:

- you can keep track of the "depth" of each list node (i.e., how far down the list it appears);
- you can maintain the abstract sequence of list nodes (i.e., a map from \natural to nodes, along with a \natural giving the length of the sequence);
- you can maintain the "reachability" relationship between ordered pairs of nodes.

For this example, we'll use the first approach. We add the following to the definition of the List type:

```
  _(ghost \natural idx[Node *])
  _(invariant head ==> idx[head] == 0)
  _(invariant \forall Node *n, *m; \mine(n) && \mine(m) ==>
      head
      && (idx[n] == 0 ==> n==head)
      && (n->nxt ==> idx[n->nxt] == idx[n] + 1)
      && (idx[m] == idx[n] + 1 ==> m == n->nxt)
      && (idx[m] > idx[n] ==> n->nxt)
  )
```

The new invariants say that the head (if it exists) is at depth 0 (and is the only node at depth 0), that depth increases by 1 when following list pointers, and that you can never "miss" a node by following the list. (These are similar to the invariants you would use if each node had a key and you were maintaining an ordered list.)

The only change to the code we've previously seen is that when adding a node to the list, we have to also update the node indices:

```
l->idx = (\lambda Node *m; m == n ? 0 : l->idx[m] + 1);
```

We can now write and verify the membership test:

```
int member(List *l, int k)
  _(requires \wrapped(l))
  // partial specification, ==> instead of <==>
  _(ensures \result == l->val[k])
{
  Node *n;
  _(assert \inv(l))
  for (n = l->head; n; n = n->nxt)
    _(invariant n ==> n \in l->\owns && n \in \domain(l))
    _(invariant \forall Node *m; m \in l->\owns && m->data == k
      ==> m==n || (n && l->idx[m] > l->idx[n]))
  {
        if (n->data == k) return 1;
  }
  _(assert l->val[k] ==> l->find[k] \in l->\owns)
  return 0;
}
```

Note that the second invariant of the loop is analogous to the invariant we would use for linear search in an array.

The _(assert) is an example of a situation where VCC needs a little bit to see why what you think is true really is true. The loop invariant says that there are no nodes in the list with key k, but VCC on its own will fail to make the appropriate connection to l->val[k] via l->find[k] without this hint (which is just giving an instantiation for the last list invariant).

Exercises

1. Modify the list implementation so that on a successful membership test, the node that is found is moved to the front of the list. (Note that the resulting function is no longer pure.)
2. Implement sets using sorted lists.
3. Implement sets using binary search trees.

6.2 Sorting revisited

In § 4.2 we verified that bubblesort returns a sorted array. But we didn't prove that it returned a permutation of the input array[23]. To express this postcondition, we return a ghost map, which states the exact permutation that the sorting algorithm produced:

```
_(typedef _(record) struct Perm {
        \natural fwd[\natural];
        \natural bwd[\natural];
} Perm;)

_(def \bool isPerm(Perm p, \natural len) {
        return (\forall \natural i; i < len ==> p.fwd[i] < len &&
                p.bwd[i] < len)
                  && (\forall \natural i; i < len ==> p.fwd[p.bwd[i]] == i
                  && p.bwd[p.fwd[i]] == i);
})

_(def Perm permId() {
        Perm p;
        p.fwd = (\lambda \natural i; i);
        p.bwd = (\lambda \natural i; i);
        return p;
})

_(def Perm permSwap(\natural i, \natural j) {
        Perm p;
        p.fwd = (\lambda \natural k; k == i ? j : (k == j ? i : k));
        p.bwd = p.fwd;
        return p;
})

_(def Perm permCompose(Perm p, Perm q) {
        Perm r;
        r.fwd = (\lambda \natural k; p.fwd[q.fwd[k]]);
        r.bwd = (\lambda \natural k; q.bwd[p.bwd[k]]);
        return r;
})

void sort(int *buf, unsigned len _(out Perm p))
  _(requires \mutable_array(buf,len))
  _(writes \array_range(buf, len))
  _(ensures sorted(buf, len))
  _(ensures isPerm(p,len))
  _(ensures \forall unsigned i; i < len ==> buf[i] ==
        \old(buf[p.fwd[i]]))
  _(decreases 0)
{
  _(ghost int av[\natural] = \lambda \natural i; buf[i])
  _(ghost p = permId())

  if (len < 2) return;

  for (unsigned i = len; i > 0; i--)
    _(invariant i <= len)
    _(invariant \forall unsigned u,v; i <= v && u <= v && v < len
          ==> buf[u] <= buf[v])
    _(invariant isPerm(p,len))
    _(invariant \forall unsigned i; i < len ==> buf[i] == av[p.fwd[i]])
    for (unsigned j = 0; j + 1 < i; j++)
      _(decreases i-j)
      _(invariant j < i)
      _(invariant \forall unsigned u,v; i <= v && u <= v && v < len
            ==> buf[u] <= buf[v])
      _(invariant \forall unsigned u; u < j ==> buf[u] <= buf[j])
      _(invariant isPerm(p,len))
      _(invariant \forall unsigned i; i < len ==> buf[i] ==
            av[p.fwd[i]])
      if (buf[j] > buf[j+1]) {
        int tmp = buf[j];
        buf[j] = buf[j+1];
        buf[j+1] = tmp;
        _(ghost p = permCompose(p,permSwap(j,j+1)))
      }
}
```

```
Verification of sorted succeeded.
Verification of isPerm succeeded.
Verification of permId succeeded.
Verification of permSwap succeeded.
Verification of permCompose succeeded.
Verification of sort succeeded.
```

This sample introduces two new features. The first is the output ghost parameter _(out Perm p). An out parameter is used to return data from the function to the caller. (One could also do it with a pointer to ghost memory, but using an out parameter is simpler and more efficient.)

To call sort() you need to supply a local variable to hold the permutation when the function exits, as in:

```
void f(int *buf, unsigned len)
  //...
{
  _(ghost Perm myperm; )
  //...
  sort(buf, len _(out myperm));
}
```

The effect is to copy the value of the local variable p of the function to the variable myperm on exit from the function.

The second feature is the use of ***record*** types. A record type is introduced by putting _(record) before the definition of a like a struct type. They are mathematically cleaner than structs, in several ways:

- A record is a single, indivisible value (like a map), so you don't have to worry about aliasing individual fields of a record.
- Two records are equal iff their corresponding fields are equal. Conversely, C doesn't allow == on struct values (primarily because padding makes the meaning problematic).
- Because == makes sense on records, a record type can be used as the domain of a map, whereas a struct type cannot.

However, records also have some limitations relative to structs:

- Records are values, not objects, so records cannot have invariants.
- Record fields can be of record type, but cannot be of compound (struct or union) type.
- You can't take the address of a field of a record (and so cannot pass it to a function that updates it). However, you can use record fields as _(out) parameters.

[23] For arrays in which no value occurs more than once, this property can be expressed that every value in the output array is in the input array. But with multiple occurrances, this would require stating that the multiplicity of each value is the same, a property that isn't first-order.

6.3 Inductive Proofs

Sometimes, you will want to verify programs that depend on mathematics that is too hard for VCC to do on its own, typically because they require guidance. In VCC, you do this by writing ghost code. In particular, you can do inductive proofs by writing loops or by writing recursive functions, with the loop invariant or function spec serving as the inductive hypothesis. Because C doesn't allow the definition of anonymous recursive functions, loops are usually more convenient.

Here is a small example of using ghost code to prove the formula for triangular numbers:

```
_(void triangle()
        _(decreases 0)
{
  \natural x[\natural];
  \natural n;
  _(assume x[0] == 0 && \forall \natural i; x[i+1] == x[i] + i + 1)
  _(ghost
      for (\natural i = 0; i < n; i=i+1)
        _(invariant i <= n && x[i] == i * (i + 1) / 2)
      {})
  _(assert x[n] == n * (n + 1) / 2)
})
```

Sometimes, you might want to use inductively defined types other than \natural or \integer. VCC lets you define your own (ghost) inductive types, much like modern functional languages, but using the following C-like syntax:

```
_(datatype List {
    case nil();
    case cons(int v, List l);
})
```

This defines an inductive datatype of lists, where a list is either of the empty list nil() or an int followed by a list. Values of abstract types are deconstructed using the following switch construct (corresponding to matching expressions in functional languages):

```
_(def List app(List x, List y) {
  switch(x) {
    case nil(): return y;
    case cons(v,l): return cons(v, app(l,y));
  }
})
```

Note that unlike the usual C switch statement, fallthrough from one case to the next is not allowed (since the scope of variables like v and l introduced by the case construct only go to the end of the case). Note that VCC automatically chooses the termination measure _(decreases size(x), size(y)), which suffices for typical functions where termination follows by structural induction.

We can now prove something about the function we defined:

```
_(def void appAssoc(List x, List y, List z)
_(ensures app(x,app(y,z)) == app(app(x,y),z))
{
  switch (x) {
    case nil(): return;
    case cons(v,l): appAssoc(l,y,z); return;
  }
})
```

Note that the recursive call provides the needed inductive case of the theorem we're trying to prove. We can similarly use the resulting function as a lemma in proving other theorems.

Exercises

1. Verify your favorite sorting functions (quicksort, heapsort, mergesort, etc.).
2. Using app, define a recursive function that reverses a List. Prove that reversing a list twice gives you the same list back. (Hint: you will need some additional lemmas along the way; one solution can be found in 7.5.lists.c.)

7. Objects and Memory

In VCC, the state of the world is given by the state of a set of completely independent ***objects***, each with a set of fields and a value for each of these fields. (That's right, the objects are completely separate.) For a given program, the set of objects is fixed. Each field of each object is either ghost or concrete; each concrete field of each object has a fixed address and size. Each object has a (volatile, ghost) Boolean field \valid that says whether the object is currently "active"; a basic invariant guaranteed by succesful verification is that concrete fields of valid objects do not overlap. Moreover, another basic invariant guarantees that threads read and write only through valid objects. These invariants guarantee that reads and writes of concrete fields can be implemented by reads and writes in the concrete address space.

The objects mostly correspond to instances of user-defined compound (i.e., struct or union) types. In addition, there are a few special objects; the most important of these are the following:

- For every variable of primitive type, there is a dummy object of which the variable is the only field.
- Each thread is an object. The thread running the current function activation is called \me, and it owns all of the \wrapped and \mutable variables.

In particular, there are no objects of primitive types (integral types or pointer types); a C "object" of primitive type is always a field of a a VCC object. If p is a pointer to a primitive lvalue, the object of which p is a field is denoted \embedding(p). The embedding of a field is considered part of its type, so pointers to two fields of different objects that happen to have the same address are not considered equal to VCC, even though they "test" as equal in C. For example,

```
void test(int *x, int *y) {
  if (x == y) {
    _(assert \addr(x) == \addr(y)) // succeeds
    _(assert x == y) // fails
  }
}
```

Note that for pointers to fields of valid objects, this discrepency disappears (since such fields cannot alias), so the discrepency in the interpreation of pointer equality rarely comes up in practice. Note also that this discrepency doesn't effect the soundness of verification, just how assertions are interpreted. If for some reason you really need to talk about raw addresses, you can always use the \addr operator.

7.1 Unions

A C union defines a group of objects that share memory addresses. Thus, the members of a union cannot in general all be valid at the same time. When VCC allocates a union, it chooses an arbitrary member to be the valid one. You can invalidate the active one and validate another using the operator union_reinterpret, as in the following example:

```
typedef struct S {
        int x;
} S;

typedef struct T {
        int y,z;
} T;
```

```
typedef union U {
        S s;
        T t;
} U;

void test() {
  U u;
  //u.s.x = 1; // fails
  _(union_reinterpret &u.s)
  u.s.x = 1;
  //u.t.y == 1; // fails
  _(union_reinterpret &u.t)
  u.t.y = 2;
}
```

7.2 Blobs

In addition to unions, on rare occasions one needs a chunk of memory that can be used to make objects of more arbitrary types. In practice, this arises primarily when you are implementing your own memory manager (typically to more efficiently manage the allocation of small objects within a page of raw memory).

In VCC, a chunk of truly "raw" memory is called a ***blob***. A blob of appropriate size and alignment can be turned into an object (along with its extent), making the blob invalid and the objects valid (and mutable). The top object of such a hierarchy is said to be blobifiable, and can be later turned back into a blob (again, this requires that its extent is mutable). Contiguous blobs can be combined into a single blob or vice versa.

When a typed object is allocated (on the heap of the stack), what is actually allocated is a blob, and this blob is unblobified into an object of the appropriate type. This means that you can blobify top-level local objects, as well as an object created from malloc'd memory. On the other hand, you cannot blobify a field of a struct.

8. Atomics

Writing concurrent programs is generally considered harder than writing sequential programs. Similar opinions are held about verification. However, in VCC the leap from verifying sequential programs to verifying fancy lock-free code is not that big. This is because verification in VCC is inherently based on object invariants.

Coarse-grained concurrency - where objects are protected by locks - is really no different from sequential programming. Each lock protects a single (typically fixed) object; when you acquire an exclusive lock, you obtain ownership of the object protected by the lock. You are then free to operate on tht object just as you would in sequential programming; you can unwrap it, play with it, and eventually wrap it up (checking that you restored the object invariant) and release the lock, giving ownership of the protected object back to the lock.

Fine-grained concurrency - where memory is accessed using atomic actions - is not very different from coarse-grained concurrency (except that one is obviously restricted to operations that are implemented atomically on the hardware platform). The main difference is that you do not get ownership of the object you operate on; instead, you operate on the objct by reading or writing its volatile fields while leaving the object closed.

We begin with what is probably the most important lock free algorith: the humble spin-lock. The spin-lock data-structure is very simple – it contains just a single boolean field, meant to indicate whether the spin-lock is currently acquired. However, like most concurrent objects, the important thing about a spinlock is forcing its users to "play by the rules". For example, the most important characteristic of a spinlock is that once you acquire a spinlock, you know that nobody else can acquire it until you release it; this would be broken if somebody else could release the lock. But what does it mean for "you" to release it? Does that mean your thread? How then would you cope with protocols where you intentionally transfer your right to release the spinlock to another thread?

Fortunately, we have the means at hand to represent an unforgable right: since each object has a unique owner, we can represent a unique right with ownership of an object. When a lock is acquired, the caller receives ownership of an object associated with the lock (not the lock itself, which will ultimately remain shared). To release the lock, the caller has to give this object back. This approach prevents other threads from competing for the lock, yet allows the thread to "give" his right to release the lock to another thread (or to store it in a data structure) by simply transfering ownership.

The protected object need not serve as a simple token; being an object, it can own data and have an invariant. These correspond to the data "protected" by the lock and the notion of "consistency" of this data.

```
_(volatile_owns) struct Lock {
  volatile int locked;
  _(ghost \object protected_obj;)
  _(invariant locked == 0 ==> \mine(protected_obj))
};
```

We use a ghost field to hold a reference to the object meant to be protected by this lock. If you wish to protect multiple objects with a single lock, you can make the object referenced by protected_obj own them all. The locked field is annotated with volatile. It has the usual meaning ascribed to the modifier in C (i.e., it makes the compiler assume that the environment might write to that field, outside the knowledge of the compiler). However, in practice C compilers also provide the stronger guarantee that volatile operations performed by a single thread are guaranteed to happen in program order.

VCC treats volatile fields differently from nonvolatile fields in two specific ways. First, a volatile field of an object can be modified while the object is closed, as long as the update occurs inside an explicit atomic action that preserves the object invariant. Second, volatile fields are not in sequential domains, so a thread forgets the values of these fields when it makes an impure function call (and, as we will see soon, just before an atomic action).

The attribute _(volatile_owns) means that we want the \owns set to be treated as a volatile field (i.e., we want to be able to write it while the object is closed); without this declaration, \owns sets can only be update when the object is open.

Here is the constructor for the lock:

```
void InitializeLock(struct Lock *l _(ghost \object obj))
  _(writes \span(l), obj)
  _(requires \wrapped(obj))
  _(ensures \wrapped(l) && l->protected_obj == obj)
{
  l->locked = 0;
  _(ghost {
    l->protected_obj = obj;
    l->\owns = {obj};
    _(wrap l)
  })
}
```

The parameter ob is passed as a ***ghost parameter***. The regular lock initialization function prototype does not say which object the lock is supposed to protect, but the lock invariant requires it. Thus, we introduce additional parameter for the purpose of verification. A call to the initialization will look like InitializeLock(&l _(ghost o)). Note that in order to allow VCC annotations to be erased by the preprocessor, ghost arguments are not separated with commas. The transfer of ownership of ob into the lock is exactly as in adding an object to a container data-structure, like sc_add() from § 5.3.

Now we can see how we operate on volatile fields. We will start with the simpler of the two lock operations, the release:

```
void Release(struct Lock *l)
  _(requires \wrapped(l))
  _(requires \wrapped(l->protected_obj))
  _(writes l->protected_obj)
{
  _(atomic l) {
    l->locked = 0;
    _(ghost l->\owns += l->protected_obj)
  }
}
```

First, let's look at the contract. The contract above requires the lock to be wrapped, which is hardly realistic for a lock (which of course must be shared). We will fix this problem later, but for now, we note that what we really need is a reliable way to make sure that the lock remains closed.

The preconditions on the protected object are very similar to the preconditions on the InitializeLock(). Note that the Release() does not need to mention the lock in its writes clause, this is because the write it performs is volatile. This is because the lock is not in the sequential domain of \me (since it is a volatile field), and so the thread forgets whatever it knew about the field when it calls a function.

The atomic block is similar in spirit to the unwrapping block — it allows for modifications of listed objects and checks if their invariants are preserved. The difference is that the entire update happens instantaneously from the point of view of other threads. We needed the unwrapping operation because we wanted to mark that we temporarily break the object invariants. Here, there is no point in time where other threads can observe that the invariants are broken. Invariants hold before the beginning of the atomic block (by our principal reasoning rule, § 5.1), and we check the invariant at the end of the atomic block.

For the use of an atomic action construct to be sound, VCC requires that the actions on volatile concrete data of closed objects performed by the action must appear atomic to other threads. VCC issues a warning if there is more than one volatile physical memory operation inside of an atomic block (since this is unlikely to be atomic on most architectures), but is otherwise silent, assuming that a single volatile memory access is implemented in an atomic way by the host compiler (even though this is not true for all data types on all platforms). Ultimately, it is up to the user to guarantee that the access performed is indeed atomic.

Here, the physically atomic operation is writing to the l->locked; Other possibilities include reading from a volatile field, or a performing a primitive operation supported by the hardware, like interlocked compare-and-exchange, on such a field. The block may also include any number of accesses to mutable variables, reads of thread-local variables, reads of nonvolatile fields of objects named in the atomic, reads of volatile ghost variables (in any object), and writes of volatile ghost variables of objects named in the atomic. The additional operations on concrete fields are allowed because these accesses are not racing with other threads, and the ghost operations are allowed because we can pretend that ghost updates happen immediately (without scheduler boundary).

It is not hard to see that this atomic operation preserves the invariant of the lock. However, VCC must also make the usual checks on the sequential data. To transfer ownership of l->protected_obj to the lock, we also need write permission to the object being transferred, and we need to know it is closed. For example, had we forgotten to mention l->protected_obj in the writes clause, VCC would have complained:

```
Verification of Lock#adm succeeded.
Verification of Release failed.
testcase(16,13) : error VC8510: Assertion
     'l->protected_obj is writable in call to l->\owns
     += l->protected_obj' did not verify.
```

And had we forgotten to perform the ownership transfer inside of Release(), VCC would have complained about the invariant of the lock:

```
Verification of Lock#adm succeeded.
Verification of Release failed.
testcase(15,12) : error VC8524: Assertion 'chunk locked
     == 0 ==> \mine(protected_obj) of invariant of l
     holds after atomic' did not verify.
```

Now we'll look at Acquire(). The specification is not very surprising. It requires the lock to be wrapped (again, unrealistic for a shared lock, but we will fix this later). It ensures that the calling thread will own the protected object, and moreover, that the thread didn't directly own it before. This is much like the postcondition on sc_set() function from § 5.3.

```
void Acquire(struct Lock *l)
  _(requires \wrapped(l))
  _(ensures \wrapped(l->protected_obj) &&
        \fresh(l->protected_obj))
{
  int stop = 0;

  do {
    _(atomic l) {
      stop = InterlockedCompareExchange(&l->locked, 1, 0) == 0;
      _(ghost if (stop) l->\owns -= l->protected_obj)
    }
  } while (!stop);
}
```

The InterlockedCompareAndExchange() function is a compiler instrinsic, which on the x86/x64 hardware translates to the cmpxchg assembly instruction. It takes a memory location and two values. If the memory location contains the first value, then it is replaced with the second. It returns the old value. The entire operation is performed atomically (and is also a write barrier).

VCC doesn't have all the primitives of all the C compilers predefined. You define such intrinsics for VCC by providing a body. It is seen only by VCC (it is enclosed in _(atomic_inline ...)) so that the normal compiler doesn't get confused about it.

```
_(atomic_inline)
int InterlockedCompareExchange(volatile int *Destination, int
      Exchange, int Comparand) {
  if (*Destination == Comparand) {
    *Destination = Exchange;
    return Comparand;
  } else {
    return *Destination;
  }
}
```

It is up to you to make sure that any such functions you provide indeed match the semantics provided by your compiler and platform. Such definitions may in future be provided in header files for certain compiler/platform combinations.

8.1 Using claims

The contracts of functions operating on the lock require that the lock is wrapped. This is because one can only perform atomic operations on objects that are closed. If an object is open, then the owning thread is in full control of it. However, wrapped means not

only closed, but also owned by the current thread, which defeats the purpose of the lock — it should be possible for multiple threads to compete for the lock. Let's then say, there is a thread which owns the lock. Assume some other thread t got to know that the lock is closed. How would t know that the owning thread won't unwrap (or worse yet, deallocate) the lock, just before t tries an atomic operation on the lock? The owning thread thus needs to somehow promise t that lock will stay closed. In VCC such a promise takes the form of a ***claim***. Later we'll see that claims are more powerful, but for now consider the following to be the definition of a claim:

```
_(ghost
typedef struct {
  \ptrset claimed;
  _(invariant \forall \object o; o \in claimed ==> o->\closed)
} \claim_struct, *\claim;
)
```

Thus, a claim is an object, with an invariant stating that a number of other objects (we call them ***claimed objects***) are closed. As this is stated in the invariant of the claim, it only needs to be true as long as the claim itself stays closed.

Recall that what can be written in invariants is subject to the admissibility condition, which we have seen partially explained in § 5.5. There we said that an invariant should talk only about things the object owns. But here the claim doesn't own the claimed objects, so how should the claim know the object will stay closed? In general, an admissible invariant can depend on other objects invariants always being preserved (we'll see the precise rule in § 8.3). So VCC adds an implicit invariant to all types marked with _(claimable) attribute. This invariant states that the object cannot be unwrapped when there are closed claims on it. More precisely, each claimable object keeps track of the count of outstanding claims. The number of outstanding claims on an object is stored in \claim_count field.

Now, getting back to our lock example, the trick is that there can be multiple claims claiming the lock (note that this is orthogonal to the fact that a single claim can claim multiple objects). The thread that owns the lock will need to keep track of who's using the lock. The owner won't be able to destroy the lock (which requires unwrapping it), before it makes sure there is no one using the lock. Thus, we need to add _(claimable) attribute to our lock definition, and change the contract on the functions operating on the lock. As the changes are very similar we'll only show Release().

```
void Release(struct Lock *l _(ghost \claim c))
  _(requires \wrapped(c) && \claims_object(c, l))
  _(requires l->protected_obj != c)
  _(requires \wrapped(l->protected_obj))
  _(ensures \wrapped(c))
  _(writes l->protected_obj)
{
  _(atomic c, l) {
    _(assert \by_claim(c, l->protected_obj) != c) // why do we
          need it?
    l->locked = 0;
    _(ghost l->\owns += l->protected_obj)
  }
}
```

We pass a ghost parameter holding a claim. The claim should be wrapped. The function \claims_obj(c, l) is defined to be l \in c->claimed, i.e., that the claim claims the lock. We also need to know that the claim is not the protected object, otherwise we couldn't ensure that the claim is wrapped after the call. This is the kind of weird corner case that VCC is very good catching (even if it's bogus in this context). Other than the contract, the only change is that we list the claim as parameter to the atomic block. Listing a normal object as parameter to the atomic makes VCC know you're going to modify the object. For claims, it is just a hint, that it should use this claim when trying to prove that the object is closed.

Additionally, the InitializeLock() needs to ensure l->\claim_count == 0 (i.e., no claims on freshly initialized lock). VCC even provides a syntax to say something is wrapped and has no claims: \wrapped0(l).

8.2 Creating claims

When creating (or destroying) a claim one needs to list the claimed objects. Let's have a look at an example.

```
void create_claim(struct Data *d)
  _(requires \wrapped(d))
  _(writes d)
{
  _(ghost \claim c;)
  struct Lock l;
  InitializeLock(&l _(ghost d));
  _(ghost c = \make_claim({&l}, \true);)
  Acquire(&l _(ghost c));
  Release(&l _(ghost c));
  _(ghost \destroy_claim(c, {&l}));
  _(unwrap &l)
}
```

This function tests that we can actually create a lock, create a claim on it, use the lock, and then destroy it. The InitializeLock() leaves the lock wrapped and writable by the current thread. This allows for the creation of an appropriate claim, which is then passed to Acquire() and Release(). Finally, we destroy the claim, which allows for unwrapping of the lock, and subsequently deallocating it when the function activation record is popped off the stack.

The \make_claim(...) function takes the set of objects to be claimed and a property (an invariant of the claim, we'll get to that in the next section). Let us give desugaring of \make_claim(...) for a single object in terms of the \claim_struct defined in the previous section.

```
// c = \make_claim({o}, \true) expands to
o->\claim_count += 1;
c = malloc(sizeof(\claim_struct));
c->claimed = {o};
_(wrap c);

// \destroy_claim(c, {o}) expands to
assert(o \in c->claimed);
o->\claim_count -= 1;
_(unwrap c);
free(c);
```

Because creating or destroying a claim on c assigns to c->\claim_count, it requires write access to that memory location. One way to obtain such access is getting sequential write access to c itself: in our example the lock is created on the stack and thus sequentially writable. We can thus create a claim and immediately use it. A more realistic claim management scenario is described in § 8.5.

The \true in \make_claim(...) is the claimed property (an invariant of the claim), which will be explained in the next section.

The destruction can possibly leak claim counts, i.e., one could say:

```
\destroy_claim(c, {});
```

and it would verify just fine. This avoids the need to have write access to p, but on the other hand prevents p from unwrapping forever (which might be actually fine if p is a ghost object).

8.3 Two-state invariants

Sometimes it is not only important what are the valid states of objects, but also what are the allowed *changes* to objects. For example, let's take a counter keeping track of certain operations since the beginning of the program.

```
_(claimable) struct Counter {
  volatile unsigned v;
  _(invariant v > 0)
  _(invariant v == \old(v) || v == \old(v) + 1)
};
```

Its first invariant is a plain single-state invariant – for some reason we decided to exclude zero as the valid count. The second invariant says that for any atomic update of (closed) counter, v can either stay unchanged or increment by exactly one. The syntax \old(v) is used to refer to value of v before an atomic update, and plain v is used for the value of v after the update. (Note that the argument to \old(...) can be an arbitrary expression.) That is, when checking that an atomic update preserves the invariant of a counter, we will take the state of the program right before the update, the state right after the update, and check that the invariant holds for that pair of states.

> In fact, it would be easy to prevent any changes to some field f, by saying _(invariant \old(f)== f). This is roughly what happens under the hood when a field is declared without the volatile modifier.

As we can see the single- and two-state invariants are both defined using the _(invariant ...) syntax. The single-state invariants are just two-state invariants, which do not use \old(...). However, we often need an interpretation of an object invariant in a single state S. For that we use the ***stuttering transition*** from S to S itself. VCC enforces that all invariants are ***reflexive*** that is if they hold over a transition S0, S1, then they should hold in just S1 (i.e., over S1, S1). In practice, this means that \old(...) should be only used to describe how objects change, and not what are their proper values. In particular, all invariants which do not use \old(...) are reflexive, and so are all invariants of the form \old(E)== (E)|| (P), for any expression E and condition P. On the other hand, the invariants \old(f)< 7 and x == \old(x)+ 1 are not reflexive.

Let's now discuss where can you actually rely on invariants being preserved.

```
void foo(struct Counter *n)
  _(requires \wrapped(n))
{
  int x, y;
  atomic(n) { x = n->v; }
  atomic(n) { y = n->v; }
}
```

The question is what do we know about x and y at the end of foo(). If we knew that nobody is updating n->v while foo() is running we would know x == y. This would be the case if n was unwrapped, but it is wrapped. In our case, because n is closed, other threads can update it, while foo() is running, but they will need to adhere to n's invariant. So we might guess that at end of foo() we know y == x || y == x + 1. But this is incorrect: n->v might get incremented by more than one, in several steps. The correct answer is thus x <= y. Unfortunately, in general, such properties are very difficult to deduce automatically, which is why we use plain object invariants and admissibility check to express such properties in VCC.

> An invariant is ***transitive*** if it holds over states S0, S2, provided that it holds over S0, S1 and S1, S2. Transitive invariants could be assumed over arbitrary pairs of states, provided that the object stays closed in between them. VCC does not require invariants to be transitive, though.
>
> Some invariants are naturally transitive (e.g., we could say _(invariant \old(x)<= x) in struct Counter, and it would be almost as good our current invariant). Some other invariants, especially the more complicated ones, are more difficult to make transitive. For example, an invariant on a reader-writer lock might say
>
> _(**invariant** writer_waiting ==> old(readers) >= readers)
>
> To make it transitive one needs to introduce version numbers. Some invariants describing hardware (e.g., a step of physical CPU) are impossible to make transitive.

Consider the following structure definition:

```
struct Reading {
  struct Counter *n;
  volatile unsigned r;
  _(ghost \claim c;)
  _(invariant \mine(c) && \claims_object(c, n))
  _(invariant n->v >= r)
};
```

It is meant to represent a reading from a counter. Let's consider its admissibility. It has a pointer to the counter, and a owns a claim, which claims the counter. So far, so good. It also states that the current value of the counter is no less than r. Clearly, the Reading doesn't own the counter, so our previous rule from § 5.5, which states that you can mention in your invariant everything that you own, doesn't apply. It would be tempting to extend that rule to say "everything that you own or have a claim on", but VCC actually uses a more general rule. In a nutshell, the rule says that every invariant should be preserved under changes to other objects, provided that these other objects change according to their invariants. When we look at our struct Reading, its invariant cannot be broken when its counter increments, which is the only change allowed by counters invariant. On the other hand, an invariant like r == n->v or r >= n->v could be broken by such a change. But let us proceed with somewhat more precise definitions.

First, assume that every object invariant holds when the object is not closed. This might sound counter-intuitive, but remember that closedness is controlled by a field. When that field is set to false, we want to *effectively* disable the invariant, which is the same as just forcing it to be true in that case. Alternatively, you might try to think of all objects as being closed for a while.

An atomic action, which updates state S0 into S1, is ***legal*** if and only if the invariants of objects that have changed between S0 and S1 hold over S0, S1. In other words, a legal action preservers invariants of updated objects. This should not come as a surprise: this is exactly what VCC checks for in atomic blocks.

An invariant is ***stable*** if and only if it cannot be broken by legal updates. More precisely, to prove that an invariant of p is stable, VCC needs to "simulate" an arbitrary legal update:

- Take two arbitrary states S0 and S1.
- Assume that all invariants (including p's) hold over S0, S0.
- Assume that for all objects, some fields of which are not the same in S0 and S1, their invariants hold over S0, S1.
- Assume that all fields of p are the same in S0 and S1.
- Check that invariant of p holds over S0, S1.

The first assumption comes from the fact that all invariants are reflexive. The second assumption is legality. The third assumption follows from the second (if p did change, its invariant would automatically hold).

An invariant is ***admissible*** if and only if it is stable and reflexive.

Let's see how our previous notion of admissibility relates to this one. If p owns q, then q \in p->\owns. By the third admissibility

assumption, after the simulated action p still owns q. By the rules of ownership (§ 5.1), only threads can own open objects, so we know that q is closed in both S0 and S1. Therefore non-volatile fields of q do not change between S0 and S1, and thus the invariant of p can freely talk about their values: whatever property of them was true in S0, will also be true in S1. Additionally, if q owned r before the atomic action, and the q−>\owns is non-volatile, it will keep owning r, and thus non-volatile fields of r will stay unchanged. Thus our previous notion of admissibility is a special case of this one.

Getting back to our foo() example, to deduce that x <= y, after the first read we could create a ghost Reading object, and use its invariant in the second action. While we need to say that x <= y is what's required, using a full-fledged object might seem like an overkill. Luckily, definitions of claims themselves can specify additional invariants.

The admissibility condition above is semantic: it will be checked by the theorem prover. This allows construction of the derived concepts like claims and ownership, and also escaping their limitations if needed. It is therefore the most central concept of VCC verification methodology, even if it doesn't look like much at the first sight.

8.4 Guaranteed properties in claims

When constructing a claim, you can specify additional invariants to put on the imaginary definition of the claim structure. Let's have a look at annotated version of our previous foo() function.

```
void readtwice(struct Counter *n)
  _(requires \wrapped(n))
  _(writes n)
{
  unsigned int x, y;
  _(ghost \claim r;)

  _(atomic n) {
    x = n->v;
    _(ghost r = \make_claim({n}, x <= n->v);)
  }

  _(atomic n) {
    y = n->v;
    _(assert \active_claim(r))
    _(assert x <= y)
  }
}
```

Let's give a high-level description of what's going on. Just after reading n−>v we create a claim r, which guarantees that in every state, where r is closed, the current value of n−>v is no less than the value of x at the time when r was created. Then, after reading n−>v for the second time, we tell VCC to make use of r's guaranteed property, by asserting that it is "active". This makes VCC know x <= n−>v in the current state, where also y == n−>v. From these two facts VCC can conclude that x <= y.

The general syntax for constructing a claim is:

```
_(ghost c = \make_claim(S, P))
```

We already explained, that this requires that s−>\claim_count is writable for s \in S. As for the property P, we pretend it forms the invariant of the claim. Because we're just constructing the claim, just like during regular object initialization, the invariant has to hold initially (i.e., at the moment when the claim is created, that is wrapped). Moreover, the invariant has to be admissible, under the condition that all objects in S stay closed as long as the claim itself stays closed. The claimed property cannot use \old(...), and therefore it's automatically reflexive, thus it only needs to be stable to guarantee admissibility.

But what about locals? Normally, object invariants are not allowed to reference locals. The idea is that when the claim is constructed, all the locals that the claim references are copied into imaginary fields of the claim. The fields of the claim never change, once it is created. Therefore an assignment x = UINT_MAX; in between the atomic blocks would not invalidate the claim — the claim would still refer to the old value of x. Of course, it would invalidate the final x <= y assert.

For any expression E you can use \at(\now(), E) in P in order to have the value of E be evaluated in the state when the claim is created, and stored in the field of the claim.

This copying business doesn't affect initial checking of the P, P should just hold at the point when the claim is created. It does however affect the admissibility check for P:

- Consider an arbitrary legal action, from S0 to S1.
- Assume that all invariants hold over S0, S0, including assuming P in S0.
- Assume that fields of c didn't change between S0 and S1 (in particular locals referenced by the claim are the same as at the moment of its creation).
- Assume all objects in S are closed in both S0 and S1.
- Assume that for all objects, fields of which are not the same in S0 and S1, their invariants hold over S0, S1.
- Check that P holds in S1.

To prove \active_claim(c) one needs to prove c−>\closed and that the current state is a ***full-stop*** state, i.e., state where all invariants are guaranteed to hold. Any execution state outside of an atomic block is full-stop. The state right at the beginning of an atomic block is also full-stop. The states in the middle of it (i.e., after some state updates) might not be.

Such middle-of-the-atomic states are not observable by other threads, and therefore the fact that the invariants don't hold there does not create soundness problems.

The fact that P follows from c's invariant after the construction is expressed using \claims(c, P). It is roughly equivalent to saying:

```
\forall \state s {\at(s, \active_claim(c))};
  \at(s, \active_claim(c)) ==> \at(s, P)
```

Thus, after asserting \active_claim(c) in some state s, \at(s, P) will be assumed, which means VCC will assume P, where all heap references are replaced by their values in s, and all locals are replaced by the values at the point when the claim was created.

[**TODO:** I think we need more examples about that at() business, claim admissibility checks and so forth]

8.5 Dynamic claim management

So far we have only considered the case of creating claims to wrapped objects. In real systems some resources are managed dynamically: threads ask for "handles" to resources, operate on them, and give the handles back. These handles are usually purely virtual — asking for a handle amounts to incrementing some counter. Only after all handles are given back the resource can be disposed. This is pretty much how claims work in VCC, and indeed they were modeled after this real-world scenario. Below we have an example of prototypical reference counter.

```
struct RefCnt {
  volatile unsigned cnt;
```

```
  _(ghost \object resource;)
  _(invariant \mine(resource))
  _(invariant \claimable(resource))
  _(invariant resource->\claim_count == cnt >> 1)
  _(invariant \old(cnt & 1) ==> \old(cnt) >= cnt)
};
```

Thus, a struct RefCnt owns a resource, and makes sure that the number of outstanding claims on the resource matches the physical counter stored in it. \claimable(p) means that the type of object pointed to by p was marked with _(claimable). The lowest bit is used to disable giving out of new references (this is expressed in the last invariant).

```
void init(struct RefCnt *r _(ghost \object rsc))
  _(writes \span(r), rsc)
  _(requires \wrapped0(rsc) && \claimable(rsc))
  _(ensures \wrapped(r) && r->resource == rsc)
{
  r->cnt = 0;
  _(ghost r->resource = rsc;)
  _(wrap r)
}
```

Initialization shouldn't be very surprising: \wrapped0(o) means \wrapped(o)&& o->\claim_count == 0, and thus on initialization we require a resource without any outstanding claims.

```
int try_incr(struct RefCnt *r _(ghost \claim c)
             _( out \claim ret))
  _(always c, r->\closed)
  _(ensures \result == 0 ==>
      \claims_object(ret, r->resource) && \wrapped0(ret) &&
          \fresh(ret))
{
  unsigned v, n;

  for (;;) {
    _(atomic c, r) { v = r->cnt; }
    if (v & 1) return -1;

    _(assume v <= UINT_MAX - 2)
    _(atomic c, r) {
      n = InterlockedCompareExchange(&r->cnt, v + 2, v);
      _(ghost
        if (v == n) ret = \make_claim({r->resource}, \true);)
    }

    if (v == n) return 0;
  }
}
```

First, let's have a look at the function contract. The syntax _(always c, P) is equivalent to:

```
  _(requires \wrapped(c) && \claims(c, P))
  _(ensures \wrapped(c))
```

Thus, instead of requiring \claims_obj(c, r), we require that the claim guarantees r->\closed. One way of doing this is claiming r, but another is claiming the owner of r, as we will see shortly.

As for the body, we assume our reference counter will never overflow. This clearly depends on the running time of the system and usage patterns, but in general it would be difficult to specify this, and thus we just hand-wave it.

The new thing about the body is that we make a claim on the resource, even though it's not wrapped. There are two ways of obtaining write access to p->\claim_count: either having p writable sequentially and wrapped, or in case p->\owner is a non-thread object, checking invariant of p->\owner. Thus, inside an atomic update on p->\owner (which will check the invariant of p->\owner) one can create claims on p. The same rule applies to claim destruction:

```
void decr(struct RefCnt *r _(ghost \claim c) _(ghost \claim
      handle))
  _(always c, r->\closed)
  _(requires \claims_object(handle, r->resource) &&
       \wrapped0(handle))
  _(requires c != handle)
  _(writes handle)
{
  unsigned v, n;

  for (;;)
    _(invariant \wrapped(c) && \wrapped0(handle))
  {
    _(atomic c, r) {
      v = r->cnt;
      _(assert \active_claim(handle))
      _(assert v >= 2)
    }

    _(atomic c, r) {
      n = InterlockedCompareExchange(&r->cnt, v - 2, v);
      _(ghost
        if (v == n) {
          _(ghost \destroy_claim(handle, {r->resource}));
        })
    }

    if (v == n) break;
  }
}
```

A little tricky thing here, is that we need to make use of the handle claim right after reading r->cnt. Because this claim is valid, we know that the claim count on the resource is positive and therefore (by reference counter invariant) v >= 2. Without using the handle claim to deduce it we would get a complaint about overflow in v - 2 in the second atomic block.

Finally, let's have a look at a possible use scenario of our reference counter.

```
_(claimable) struct A {
  volatile int x;
};

struct B {
  struct RefCnt rc;
  struct A a;
  _(invariant \mine(&rc))
  _(invariant rc.resource == &a)
};

void useb(struct B *b _(ghost \claim c))
  _(always c, b->\closed)
{
  _(ghost \claim ac;)
  if (try_incr(&b->rc _(ghost c) _(out ac)) == 0) {
    _(atomic &b->a, ac) {
      b->a.x = 10;
    }
    decr(&b->rc _(ghost c) _(ghost ac));
  }
}

void initb(struct B *b)
  _(writes \extent(b))
  _(ensures \wrapped(b))
{
  b->a.x = 7;
  _(wrap &b->a)
  init(&b->rc _(ghost &b->a));
  _(wrap b)
}
```

The struct B contains a struct A governed by a reference counter. It owns the reference counter, but not struct A (which is owned by the reference counter). A claim guaranteeing that struct B is closed also guarantees that its counter is closed, so we can pass it to try_incr(), which gives us a handle on struct A.

Of course a question arises where one does get a claim on struct B from? In real systems the top-level claims come either from global objects that are always closed, or from data passed when the thread is created.

9. Triggers

The triggers are likely the most difficult part of this tutorial. VCC tries to infer appropriate triggers automatically, so trigger annotations were not needed for the examples in the tutorial. However, you may need them to deal with more complex VCC verification tasks.

This appendix gives some background on the usage of triggers in the SMT solvers, the underlying VCC theorem proving technology.

SMT solvers prove that the program is correct by looking for possible counterexamples, or ***models***, where your program goes wrong (e.g., by violating an assertion). Once the solver goes through *all* possible counterexamples, and finds them all to be inconsistent (i.e., impossible), it considers the program to be correct. Normally, it would take virtually forever, for there is very large number of possible counterexamples, one per every input to the function (values stored in the heap also count as input). To workaround this problem, the SMT solver considers ***partial models***, i.e., sets of statements about the state of the program. For example, the model description may say x == 7, y > x and ∗p == 12, which describes all the concrete models, where these statements hold. There is great many such models, for example one for each different value of y and other program variables, not even mentioned in the model.

It is thus useful to think of the SMT solver as sitting there with a possible model, and trying to find out whether the model is consistent or not. For example, if the description of the model says that x > 7 and x < 3, then the solver can apply rules of arithmetic, conclude this is impossible, and move on to a next model. The SMT solvers are usually very good in finding inconsistencies in models where the statements describing them do not involve universal quantifiers. With quantifiers things tend to get a bit tricky.

For example, let's say the model description states that the two following facts are true:

```
\forall unsigned i; i < 10 ==> a[i] > 7
a[4] == 3
```

The meaning of the universal quantifier is that it should hold not matter what we substitute for i, for example the universal quantifier above implies the following facts (which are called ***instances*** of the quantifier):

```
4 < 10 ==> a[4] > 7 // for i == 4
```

which happens to be the one needed to refute our model,

```
11 < 10 ==> a[11] > 7 // for i == 11
```

which is trivially true, because false implies everything, and

```
k < 10 ==> a[k] > 7 // for i == k
```

where k is some program variable of type unsigned.

However, there is potentially infinitely many such instances, and certainly too many to enumerate them all. Still, to prove that our model candidate is indeed contradictory we only need the first one, not the other two. Once the solver adds it to the model description, it will simplify 4 < 10 to true, and then see that a[4] > 7 and a[4] == 3 cannot hold at the same time.

The question remains: how does the SMT solver decide that the first instance is useful, and the other two are not? This is done through so called ***triggers***. Triggers are either specified by the user or inferred automatically by the SMT solver or the verification tool. In all the examples before we relied on the automatic trigger inference, but as we go to more complex examples, we'll need to consider explicit trigger specification.

A trigger for a quantified formula is usually some subexpression of that formula, which contains all the variables that the formula quantifies over. For example, in the following formula:

```
\forall int i; int p[int]; is_pos(p, i) ==> f(i, p[i]) && g(i)
```

possible triggers include the following expressions is_pos(p, i), p[i], and also f(i, p[i]), whereas g(i) would not be a valid trigger, because it does not contain p.

Let's assume that is_pos(p, i) is the trigger. The basic idea is that when the SMT solvers considers a model, which mentions is_pos(q, 7) (where q is, e.g., a local variable), then the formula should be instantiated with q and 7 substituted for p and i respectively.

Note that the trigger f(i, p[i]) is ***more restrictive*** than p[i]: if the model contains f(k, q[k]) it also contains q[k]. Thus, a "bigger" trigger will cause the formula to be instantiated less often, generally leading to better proof performance (because the solver has less formulas to work on), but also possibly preventing the proof altogether (when the solver does not get the instantiation needed for the proof).

Triggers cannot contain boolean operators or the equality operator. As of the current release, arithmetic operators are allowed, but cause warnings and work unreliably, so you should avoid them.

A formula can have more than one trigger. It is enough for one trigger to match in order for the formula to be instantiated.

Multi-triggers: Consider the following formula:

```
\forall int a, b, c; P(a, b) && Q(b, c) ==> R(a, c)
```

There is no subexpression here, which would contain all the variables and not contain boolean operators. In such case we need to use a ***multi-trigger***, which is a set of expressions which together cover all variables. An example trigger here would be {P(a, b), Q(b, c)}. It means that for any model, which has both P(a, b) and Q(b, c) (for the same b!), the quantifier will be instantiated. In case a formula has multiple multi-triggers, *all* expressions in at least *one* of multi-triggers must match for the formula to be instantiated.

If it is impossible to select any single-triggers in the formula, and none are specified explicitly, Z3 will select *some* multi-trigger, which is usually not something that you want.

9.1 Matching loops

Consider a model description

```
\forall struct Node ∗n; {\mine(n)} \mine(n) ==> \mine(n−>next)
\mine(a)
```

Let's assume the SMT solver will instantiate the quantifier with a, yielding:

```
\mine(a) ==> \mine(a−>next)
```

It will now add \mine(a−>next) to the set of facts describing the model. This however will lead to instantiating the quantifier again, this time with a−>next, and in turn again with a−>next−>next and so forth. Such situation is called a ***matching loop***. The SMT solver

would usually cut such loop at a certain depth, but it might make the solver run out of time, memory, or both.

Matching loops can involve more than one quantified formula. For example consider the following, where f is a user-defined function.

```
\forall struct Node *n; {\mine(n)} \mine(n) ==> f(n)
\forall struct Node *n; {f(n)} f(n) ==> \mine(n->next)
\mine(a)
```

9.2 Trigger selection

The explicit triggers are listed in {...}, after the quantified variables. They don't have to be subexpressions of the formula. We'll see some examples of that later. When there are no triggers specified explicitly, VCC selects the triggers for you. These are always subexpressions of the quantified formula body. To select default triggers VCC first considers all subexpressions which contain all the quantified variables, and then it splits them into four categories:

- level 0 triggers, which are mostly ownership-related. These are \mine(E), E1 \in \owns(E2), and also E1 \in0 E2 (which, except for triggering, is the same as E1 \in E2).
- level 1 triggers: set membership and maps, that is expressions of the form E1 \in E2 and E1[E2].
- level 2 triggers: default, i.e., everything not mentioned elsewhere. It is mostly heap dereferences, like *p, &a[i] or a[i], as well as bitwise arithmetic operators.
- level 3 triggers: certain "bad triggers", which use internal VCC encoding functions.
- level 4 triggers: which use interpreted arithmetic operations (+, −, and * on integers).

Expressions, which contain <=, >=, <, >, ==, !=, ||, &&, ==>, <==>, and ! are not allowed in triggers.

Each of these expressions is then tested for immediate matching loop, that is VCC checks if instantiating the formula with that trigger will create a bigger instance of that very trigger. Such looping triggers are removed from their respective categories. This protects against matching loops consisting of a single quantified formula, but matching loops with multiple formulas are still possible.

To select the triggers, VCC iterates over levels, starting with 0. If there are some triggers at the current level, these triggers are selected and iteration stops. This means that, e.g., if there are set-membership triggers then heap dereference triggers will not be selected.

If there are no triggers in levels lower than 4, VCC tries to select a multi-trigger. It will only select one, with possibly low level, and some overlap between variables of the subexpressions of the trigger. Only if no multi-trigger can be found, VCC will try to use level 4 trigger. Finally, if no triggers can be inferred VCC will print a warning.

As a post-processing step, VCC looks at the set of selected triggers, and if any there are two triggers X and Y, such that X is a subexpression of Y, then Y is removed, as it would be completely redundant.

You can place a {:level N} annotation in place of a trigger. It causes VCC to use all triggers from levels 0 to N inclusive. If this results in empty trigger set, the annotation is silently ignored.

The flag `/dumptriggers:K` (or `/dt:K`) can be used to display inferred triggers. `/dt:1` prints the inferred triggers, `/dt:2` prints what triggers would be inferred if {:level ...} annotation was supplied. `/dt:3` prints the inferred triggers even when there are explicit triggers specified. It does not override the explicit triggers, it just print what would happen if you removed the explicit trigger.

Let's consider an example:

```
int *buf;
unsigned perm[unsigned];
\forall unsigned i; i < len ==> perm[i] == i ==> buf[i] < 0
```

The default algorithm will infer {perm[i]}, and with {:level 1} it will additionally select {&buf[i]}. Note the ampersand. This is because in C buf[i] is equivalent to *(&buf[i]), and thus the one with ampersand is simpler. You can also equivalently write it as {buf + i}. Note that the plus is not integer arithmetic addition, and can thus be safely used in triggers.

Another example would be:

```
\forall struct Node *n; n \in q->\owns ==> perm[n->idx] == 0
```

By default we get level 0 {n \in q->\owns}, with level 1 we also get {perm[n->idx]} and with level 2 additionally {&n->idx}.

9.3 Hints

Consider a quantified formula \forall T x; {:hint H} E. Intuitively the hint annotation states that the expression H (which can refer to x) might have something to do with proving E. A typical example, where you might need it is the following:

```
\forall struct Node *n; \mine(n) ==> \mine(n->next) &&
    n->next->prev == n
```

The default trigger selection will pick {\mine(n->next)}, which is also the "proper" trigger here. However, when proving admissibility, to know that n->next->prev did not change in the legal action, we need to know \mine(n->next). This is all good, it's stated just before, but the SMT solver might try to prove n->next->prev == n first, and thus miss the fact that \mine(n->next). Therefore, we will need to add {:hint \mine(n->next)}. For inferred level 0 triggers, these are added automatically.

References

[1] 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 *Theorem Proving in Higher Order Logics (TPHOLs 2009)*, volume 5674 of *LNCS*, pages 23–42. Springer, 2009. Invited paper.

[2] Ernie Cohen, Michał Moskal, Wolfram Schulte, and Stephan Tobies. Local verification of global invariants in concurrent programs. In Byron Cook, Paul Jackson, and Tayssir Touili, editors, *Computer Aided Verification (CAV 2010)*, volume 6174 of *Lecture Notes in Computer Science*, pages 480–494, Edinburgh, UK, July 2010. Springer.