Verifying C Programs: A VCC Tutorial

Working draft, version 0.2, May 8, 2012

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]. 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>

int main()
```

[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 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 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 VCCTutorial.sln [**TODO:** make sure that VCC can verify code not in the project] 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).

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 § 6.5). 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 loses some information when reasoning about loops, necessitating additional annotations.

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 (§ D) 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.
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.
4. Write an assertion that says that the int x is a perfect square (i.e., a number being a square of an integer).

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

[6] VCC currently doesn't check that your program doesn't run forever or run out of stack space, but future versions will, at least for sequential programs.

5. Write an assertion that says that the int x occurs in the int array b[10].
6. Write an assertion that says that the int array b, of length N, contains no duplicates.
7. Write an assertion that says that all pairs of adjacent elements of the int array b of length N differ by at most 1.
8. Write an assertion that says that an array b of length N contains only perfect squares.

2.2 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 assumptions (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 only report an error for the first assumption and not 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))
```

?

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 three types of annotations:

- A requirement on the caller (sometimes called the ***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.

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.[7]

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;
  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[8]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. Try replacing the < in the return statement of min() with >. Before running VCC, can you guess which parts of the verification will fail?
2. 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?
3. 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?
4. Specify a function that returns the (int) square root of its (int) argument. (You can try writing an implementation for the func-

[7] 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).

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

tion, but won't be able to verify it until you've learned about loop invariants.)

5. Can you think of useful functions in a real program that might legitimately guarantee only the trivial postcondition _(ensures \true)?

3.1 Side Effects

[**TODO:** move writes clauses in here, use copy example]

3.2 Reading and Writing Memory

[**TODO:** add a note about stack variables]

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, the object must reside at a valid memory address[9]. (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); in VCC, this condition is written \thread_local(p), where p is a pointer to the memory object. VCC asserts this before any sequential memory access to the object.

The three simple ways of specifying this are as follows: [**TODO:** There are some problems here. First, we have unnecessarily tangled up the classification of memory with the specification of functions by saying that they have to be in preconditions; they should instead be presented as assertions. Second, as far as i can find, we never define thread locality anywhere, and define mutability only in passing; we should give at least intuitive meanings here. Third, saying that a pointer is considered thread-local when you specify that it is thread-local just sounds vacuously stupid.-E]

- A pointer p is considered ***thread-local*** when you specify:

  _(requires \thread_local(p))

  When reading ∗p (without additional annotations) you will need to prove \thread_local(p).

- A pointer p is considered ***mutable*** when you specify:

  _(requires \mutable(p))

  All mutable pointers are also thread local. Writing via pointers different than p cannot make p non-mutable.

- A pointer p is considered ***writable*** when you specify: [**TODO:** We should introduce writability as a predicate.]

  _(writes p)

  Additionally, freshly allocated pointers are are also writable. All writable pointers are also mutable. To write through p you will need to prove that the pointer is writable. Deallocating p requires that it is writable, and afterwards the memory is not even thread-local anymore.

[9] VCC actually uses a somewhat stronger validity condition because of its typed view of memory § **??**.

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 § 6.2, 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).

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. VCC allows you to write 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)
```

3.3 Arrays

[**TODO:** should the inline expressions below be displayed like the writes clauses?] 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.

[**TODO:** Should we talk about termination here?]

3.4 Logic functions

[**TODO:** These should probably be put into an appendix, or just left out altogeather in favor of pure functions. It certainly could be delayed, right?] Just like with programs, as the specifications get more complex, it's good to structure them somewhat. One mechanism provided by VCC for that is ***logic functions***. They essentially work as macros for pieces of specification, but prevent name capture and give better error messages. For example:

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

A partial spec for a sorting routine could look like the following:[10]

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

Logic functions are not executable, but they can be implemented:

```
int check_sorted(int *arr, unsigned len)
  _(requires \thread_local_array(arr, len))
  _(ensures \result == sorted(arr, len))
{
  if (len <= 1)
    return 1;
  if (!(arr[len − 2] <= arr[len − 1]))
    return 0;
  return check_sorted(arr, len − 1);
}
```

A logic function and its C implementation can be combined into one using _(pure) annotation. This is covered in § E.1. [**TODO:** This isn't right, is it? Logic functions really act as macros, e.g. they cannot be used in triggers.]

1. Write and verify a program that checks whether two sorted arrays contain a common element.
2. Write and verify a program for binary search (a program that checks whether a sorted array contains a given value).
3. Write and verify a program that takes a 2-dimensional array of ints in which every row and column is sorted, and checks whether a given int occurs in the array.

Exercises

In the following exercises,all implementations should be recursive. You should return to these and write/verify iterative implementations after reading section § **??**.

[10] We will take care about input being permutation of the output in § 7.

1. Could the postcondition of copy have been written equivalently as the simpler ∗to == ∗from? If not, why not?
2. Specify and verify a program that takes two arrays and checks whether the arrays are equal (i.e., whether they contain the same sequence of elements).
3. Specify and verify a program that takes an array and checks whether it contains any duplicate elements. Verify a recursive implementation.
4. Specify and verify a program that checks whether a given array is a palindrome.
5. Specify and verify a program that checks whether two arrays contain a common element.
6. Specify and verify a function that swaps the values pointed to by two int pointers. Hint: use \old(...) in the postcondition.
7. Specify and verify a function that takes a text (stored in an array) and a string (stored in an array) and checks whether the string occurs within the text.
8. Extend the specification of sorting to guarantee that the sort is stable (i.e., specify that it shouldn't change the array if it is already sorted).
9. Extend the specification of sorting to guarantee that the sorting doesn't change the set of numbers occurring in the array (though it might change their multiplicities).
10. Write, without using the % operator, logic functions that specify (1) that one number (unsigned int) divides another evenly, (2) that a number is prime, (3) that two numbers are relatively prime, and (4) the greatest common divisor of two numbers.

[**TODO:** if this section remains after the functions section, move the exercises from there to here, since they require quantification.]

4. Arithmetic and Quantifiers

[**TODO:** move arithmetic stuff appendix in, as well as mathint and a mention of maps] [**TODO:** add appendix section gathering together all annotations and extensions to C]

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***.
- VCC also provides some mathematical types that cannot be used in ordinary C code (because they are too big to fit in memory); these include mathematical (unbounded) integers and (possibly infinite) maps. They are described in § **??**.
- 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 (§ E.1).

5. Loop invariants

For the most part, what VCC knows at a control point can be computed from what it knew at the immediately preceding control points. 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 a ***loop invariant***. To make sure that the loop invariant does indeed hold whenever control reaches the top of the loop, VCC asserts that the invariant holds 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;
}
```

```
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)

  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),[11] 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;
}
```

[11] Because of aliasing, it is not always obvious to VCC that a variable is not modified in the body of the loop. However, VCC can check it syntactically for a local variable if you never take the address of that variable.

```
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].

5.1 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. Let's see an example of such implicit writes clause, a reinterpretation of my_memcpy() from § **??**.

```
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]))
{
  unsigned k;
  for (k = 0; k < len; ++k)
    _(invariant \forall unsigned i; i < k ==> dst[i] == \old(src[i]))
  {
    dst[k] = src[k];
  }
}
```

If the 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)
{
  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.

[**TODO:** one sorting example is needed if we want people to do other, maybe we should use bubble or insertion sort though] Equipped with that knowledge we can proceed to not only checking if an array is sorted, as we did in § **??**, but to actually sorting it. The function below implements the bozo-sort algorithm. The algorithm works by swapping two random elements in an array, checking if the resulting array is sorted, and repeating otherwise. We do not recommend using it in production code: it's not stable, and moreover has a fairly bad time complexity.

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

void bozo_sort(int *buf, unsigned len)
  _(writes \array_range(buf, len))
  _(ensures sorted(buf, len))
{
  if (len == 0) return;

  for (;;)
    _(invariant \mutable_array(buf, len))
  {
    int tmp;
    unsigned i = random(len), j = random(len);

    tmp = buf[i];
    buf[i] = buf[j];
    buf[j] = tmp;

    for (i = 0; i < len − 1; ++i)
      _(invariant sorted(buf, i + 1))
    {
      if (buf[i] > buf[i + 1]) break;
    }

    if (i == len − 1) break;
  }
}
```

The specification that we use is that the output of the sorting routine is sorted. Unfortunately, we do not say that it's actually a permutation of the input. We'll show how to do that in § 7.2.

Exercises

Return to the verification exercises of section § 3.4 and repeat them with iterative implementations.

6. Object invariants

Pre- and postconditions allow for associating consistency conditions with the code. However, fairly often it is also possible to associate such consistency conditions with the data and require all the code operating on such data to obey the conditions. As we will learn in § 8 this is particularly important for data accessed concurrently by multiple threads, but even for sequential programs enforcing consistency conditions on data reduces annotation clutter and allows for introduction of abstraction boundaries.

In VCC, the mechanism for enforcing data consistency is ***object invariants***, which are conditions associated with compound C types (structs and unions). The invariants of a type describe how "proper" objects of that type behave. In this and the following section, we consider only the static aspects of this behavior, namely what the "consistent" states of an object are. Dynamic aspects, i.e., how objects can change, are covered in § 8. For 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 consistent SafeStrings have length not more than SSTR_MAXLEN and are '\0'-terminated. Within a type invariant, \this refers to (the address of) 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 nontrivial object invariant could be enforced to hold at all times (they would not hold right after allocation). ***Wrapped*** objects are ones for which the invariant holds and which the current thread directly owns (that is they are not part of representation of some higher-level objects). After allocating an object we would usually wrap it to make sure its invariant holds and prepare it for later use:

```
void sstr_init(struct SafeString *s)
  _(writes \span(s))
  _(ensures \wrapped(s))
{
  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 any of the assignment statements.)

A function that modifies a wrapped object will first unwrap it, make the necessary updates, and wrap the object again (which causes another check of the object invariant). 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)
{
  _(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 will 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)
{
  unsigned i;
  for (i = 0; i < s−>len; ++i)
    if (s−>content[i] == c) return (int)i;
  return −1;
}
```

The following subsection explains this wrap/unwrap protocol in more details.

6.1 Wrap/unwrap protocol

Because invariants do not always hold, in VCC one needs to explicitly state which objects are consistent, using a field \closed which is defined on every object. A ***closed object*** is one for which the \closed field is true, and an ***open object*** is one where it is false. The invariants have to (VCC enforces them) to hold only when for closed objects, but can also hold for open objects. Newly allocated objects are open, and you need to make them open before disposing them.

In addition to the \closed field each object has an ***owner field***. The owner of p is p−>\owner. This field is of pointer (object) type, but VCC provides objects, of \thread type, to represent threads of execution, so that threads can also own objects. The idea is that the owner of p should have some special rights to p that others do not. In particular, the owner of p can transfer ownership of p to another object (e.g., a thread can transfer ownership of p from itself to the memory allocator, in order to dispose of p).

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.

(Some of) the rules of ownership and consistency are

1. on every atomic step of the program the invariants of all the closed objects have to hold,
2. only the owning thread can modify fields of an open object,
3. each thread owns itself, and
4. only threads can own open objects.

Thus, by the first two rules, VCC allows updates of objects in the following two situations:

1. the updated object is closed, the update is atomic, and the update preserves the invariant of the object,
2. or the updated object is open and the update is performed by the owning thread.

In the first case to ensure that an update is atomic, VCC requires that the updated field has a volatile modifier. There is a lot to be said about atomic updates in VCC, and we shall do that in § 8, but for now we're only considering sequentially accessed objects, with no volatile modifiers on fields. For such objects we can assume that they *do not change* when they are closed, so the only way to change their fields is to first make them open, i.e., via method 2 above.

A thread needs to make the object open to update it. Because making it open counts as an update, the thread needs to own it first. This is performed by the unwrap operation, which translates to the following steps:

1. assert that the object is in the writes set,
2. assert that the object is wrapped (closed and owned by the current thread),
3. assume the invariant (as a consequence of rule 1, the invariant holds for every closed object),
4. set the \closed field to false, and
5. add the span of the object (i.e., all its fields) to the writes set

The wrap operation does just the reverse:

1. assert that the object is mutable (open and owned by the current thread),
2. assert the invariant, and
3. set the \closed field to true (this implicitly prevents further writes to the fields of the object).

Let's then have a look at the definitions of \wrapped(...) and \mutable(...).

```
logic bool \wrapped(\object o) =
  o−>\closed && o−>\owner == \me;
```

```
logic bool \mutable(\object o) =
  !o−>\closed && o−>\owner == \me;
```

The definitions of \wrapped(...) and \mutable(...) use the \object type. It is much like void∗, in the sense that it is a wildcard for any pointer type. However, unlike void∗, it also carries the dynamic information about the type of the pointer. It can be only used in specifications.

The assert/assume desugaring of the sstr_append_char() function looks 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)
}
```

6.2 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. In real programs such abstraction become hierarchical, e.g., an address book might consists of a few hash tables, each of which consists of a control object, an array of buckets, and the attached linked lists.

```
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 a variable). Such types can be only used in casts. For example, (char[capacity])content means to take the pointer content and interpret it as an array of capacity elements of type char. This notation is used so we can think of arrays as objects (of a special type). The other way to think about it is that content represents just one object of type char, whereas (char[capacity])content is an object representing the array.

The invariant of SafeString specifies that it ***owns*** the array object. The syntax \mine(o1, ..., oN) is roughly equivalent (we'll get into details later) to:

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

Conceptually there isn't much difference between having the char array embedded and owning a pointer to it. In particular, the functions operating on some s of type SafeString should still list only s in their writes clauses, and not also (char[s−>capacity])s−>content, or any other objects the string might comprise of. To achieve that VCC performs ***ownership transfers***, i.e., assignments to the \owner field. Specifically, there is another step when unwrapping an object p:

6. for each object o owned by p, set o−>\owner to \me and add o to the writes set

Similarly, when wrapping p, VCC additionally does:

4. for each object o that needs to be owned by p (which is determined by p's invariant, as you'll see in the next section), assert that o is wrapped and writable and set o−>\owner to p.

Let's have a look at an example:

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

First, let's explain the syntax:

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

is equivalent to:

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

[**TODO:** should we use "s" instead of "the string", and similarly for content?] Thus, at the beginning of the function the string is owned by the current thread and closed (i.e., wrapped), whereas, by the string invariant, the content is owned by the string and closed. After unwrapping the string, the ownership of the content goes to the current thread, but the content remains closed. Thus, unwrapping the string makes the string mutable, and the content wrapped. Then we unwrap the content (which doesn't own anything, so the thread gets no new wrapped objects), perform the changes, and wrap the content. Finally, we wrap the string. This transfers ownership of the content from the current thread to the string, so the content is no longer wrapped (but still closed). Second, let's have a look at 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; )
}
```

To make it easier to read, we made it store the s−>content pointer casted to an array into a temporary variable. Also, an invariant of p is referred to as \inv(p). As you can see there are two ownership transfers of cont to and from \me. This happens because s owns cont beforehand, as specified in its invariant. However, let's say 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, should we transfer ownership of a, b, or both? By default VCC will reject such invariants, and only allow \mine(...) as a top-level conjunct in an invariant. Invariants like the ones above are supported, but need additional annotation and manual ownership transfer when wrapping, see § 6.3.

6.3 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 states that there are no duplicates in that array. Let's now say we want to change a pointer in that array, from x to y. After such an operation, the container should own whatever it used to own minus x plus y. To facilitate such transfers VCC introduces the ***owns set***. It is essentially the inverse of the owner field. It is defined on every object p and referred to as p−>\owns. VCC maintains that:

```
\forall \object p, q; p−>\closed ==>
  (q \in p−>\owns <==> q−>\owner == p)
```

The operator <==> reads "if and only if", and is simply boolean equality (or implication both ways), with a binding priority lower than implication. That is, for closed p, the set p−>\owns contains exactly the objects that have p as their owner. Additionally, the unwrap operation does not touch the owns set, that is after unwrapping p, the p−>\owns still contains all that objects that p used to own. Finally, the wrap operation will attempt to transfer ownership of everything in the owns set to the object being wrapped. This requires that the current thread has write access to these objects and that they are wrapped.

Thus, the usual pattern is to unwrap the object, potentially modify the owns set, and wrap the object. Note that when no ownership transfers are needed, one can just unwrap and wrap the object, without worrying about ownership. Let's have a look at an example, which does perform 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)
{
  _(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, i.e., owned by the current thread.

[**TODO:** We should have entire section about BVD. –MM] VCC needs a help in form of an assertion statement at the beginning: sc_set gets a wrapped c and s, so it cannot be the case that c owns s. This is what the assertion says. Without spelling it out explicitly, VCC thinks that s might be somewhere in the strings array beforehand, and thus after inserting it at idx the distinctness invariant could be violated. If you look at the error model in that case, you can see that VCC knows nothing about the truth value of s \in c->\owns, and thus adding an explicit assertion about it helps.

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.

How is the write set updated?

Before allowing a write to *p VCC will assert \mutable(p). Additionally, it will assert that either p is in the writes clause, or the consistency or ownership of p was updated after the current function started executing. Thus, after you unwrap an object, you modify consistency of all its fields, which provides the write access to them. Also, you modify ownership of all the objects that it used to own, providing write access to unwrap these objects. In case a write clause is specified on a loop, think of an implicit function definition around the loop.

6.4 Ownership domains

An ***ownership domain*** of an object p is the set of objects that it owns and their ownership domains, plus p itself. In other words, it's the set of objects that are transitively owned by a given object.

[**TODO:** this remark might be confusing, and possibly no longer true] In general there can be cycles in the ownership graph, and so the definition above should be understood in the least fix point sense. However, every object has exactly one owner and threads own themselves, and thus anything that is owned by a thread will have a ownership domain that is a tree. It is most useful to think about ownership as trees, and disregard the degenerate cycle case.

Let's then take a look at an ownership graph: we will have a bunch of threads as roots. In particular, the current thread will own a number of objects, some of them mutable (open and thus not owning anything), but other wrapped (closed and so with possibly large ownership trees (domains) hanging off them). The ownership domains of the wrapped objects are disjoint (in the grand ownership tree of the thread they are all at the same level).

Mentioning a wrapped object in the writes clause gives the function a right to unwrap it, and then unwrap everything it owns. Thus, it effectively gives write access to its entire ownership domain. [**TODO:** make it a real example that does something useful] Consider the following piece of code:

```
void f(T *p)
  _(writes p) { ... }
...
T *p, *q, *r;
_(assert \wrapped(p) && \wrapped(q) && p != q);
_(assert q \in \domain(q))
_(assert r \in \domain(q))
_(assert q->f == 1 && r->f == 2);
f(p);
_(assert q->f == 1 && r->f == 2);
```

The function \domain(o) returns the ownership domain of o. We have three objects, two of them are wrapped. We call a function that will update one of them. We now want to know if the values of the other two are preserved. Clearly, because p != q and both are wrapped, then q is not in the ownership domain of p, so value of q->f should be preserved by the call. The value of r->f will be preserved unless r \in \domain(p), because f(p) could have written everything in \domain(p) (according to its writes clause). Unfortunately, the underlying logic used by VCC is not strong enough to show this directly (technically: the transitive closure of a relation, ownership in this case, is not expressible in first-order logic). However, VCC knows that there is no way to change anything in an ownership domain without writing its root. This is because we always enforce that ownership domains are disjoint. For example, the only way for f(p) to write something in \domain(q) would be to list q in f()'s writes clause, which is not the case. Thus, if VCC knows that r \in \domain(q), and it knows that f(p) couldn't have written q, then it also knows that r->f is unchanged. Unfortunately, we need to explicitly tell VCC in which ownership domain r is to make use of that. This is what the assertion r \in \domain(q) is doing. We currently also need to assert q \in \domain(q) to help VCC with reasoning. This is because it treats the fields of q similarly to objects owned by q. We plan to fix that in future.

6.5 Simple sequential admissibility

Until now we've been skimming on the issue of what you can actually mention in an invariant. Intuitively the invariant of an object should talk about closed states of that very object, not some other objects. However, for example the invariant of the struct SafeString talks about the values stored in its underlying array. This also seems natural: one should be able to mention things from the ownership domain of p in p's invariant.

This is important, because VCC checks only invariants of objects that you actually modify, and as we recall the most important verification property we want to enforce (and which we rely on in our verifications) is that all invariant of all (closed) objects are always preserved (§ 6.1). For example, consider:

```
struct A { int x; };
struct A *a; // global variable
struct B {
  int y;
  _(invariant a->x == y)
};
void foo()
  _(requires \wrapped(a))
  _(writes a)
{
  _(unwrapping a) { a->x = 7; }
}
```

In foo(), when wrapping a we would only check invariant of a, not all struct Bs that could possibly depend on it. Thus, an action which preserves invariant of modified object breaks invariant of another object. For this reason VCC makes the invariant of struct B inadmissible. In fact, for all invariants VCC will check that they are admissible. Admissibility of type T is checked by verifying VCC-generated function called T#adm. You can see messages about them when you verify files with type invariants.

We shall refrain now from giving a full definition of admissibility, as it only makes full sense after we learn about two-state object invariants (see § 8.3), but for sequential programs the useful approximation is that invariants that only talk about their ownership domains are admissible.

6.6 Type safety

Throughout this section we have been talking about typed memory "objects" as if this were a meaningful concept. This typed view of memory is supported by most modern programming languages, like Java, C#, and ML, where memory consists of a collection of typed objects. Programs in these languages don't allocate memory (on the stack or on the heap), they allocate objects, 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 memory, or even having two instances of the same struct type partially overlapping. Moreover, a non-null pointer might still point into an invalid region of memory.

That said, most C functions 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 objects 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, VCC essentially maintains a typed view of memory; in any state, p->\valid means that p points to memory that is currently "has" type p. The rules governing validity guarantee that in any state, the valid pointers constitute a typesafe view of memory. In particular, if two valid pointers point to overlapping portions of memory, one of them is properly contained in the other; if a struct is typed, then each of its members is typed; and if a union is typed, then exactly one of its members is typed. The definition of validity is folded into the definitions of \thread_local and \mutable; these definitions check not only that the memory pointed to exists, but that the pointer to it is valid.

There are rare situations where a program needs to change type assignment of a pointer. The most common 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 § B.1) that explicitly change the typestate. Thus, while your program can access memory using pretty much arbitrary types and typecasting, doing so will require additional annotations. But for most programs, checking type safety is completely transparent, so you don't have to worry about it.

7. Ghosts

Usually there are many ways of implementing a given data structure. For example, a set might be implemented as a linked list, an array, or a hash table.

When reasoning about a program which uses a data structure we don't want to be concerned with implementation details of the data structure. We should reason at a somewhat higher, abstract level. For example, when we use a linked list as a representation of a set, we should not be concerned with how the list nodes are laid out in memory. You would then need to include ***ghost data*** to store this set and write small bits of ***ghost code*** to update it. Ghost code is seen by VCC but not by the C compiler, and so introduces no runtime overhead. Part of the VCC philosophy is that programmers would rather do extra programming than drive interactive theorem provers, so ghost code is the preferred way to help VCC understand why your program works.

In fact we have already used ghost data, which VCC introduced: \closed, \owner, and friends are all ghost fields, not seen by the C compiler. The _(wrap ...) and _(unwrap ...) operations are ghost code. In this section we're just going to introduce some ghost data and code ourselves.

It is only rarely the case that a simple C type, say unsigned int, would be suitable to store such a data-structure abstraction (how would one store an unbounded set in a primitive C type?). To that end, VCC provides ***map types***. The syntax is similar to syntax of array types, int m[T*] defines a map m from T* to int. That is the type of expression m[p] is int, provided that p is a pointer of type T*. A map T* a[unsigned] is similar to an array of pointers of length 2^{32}.[12] 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.

Let's then have a look at an example of a list abstracted as a set:

```
struct Node {
  struct Node *next;
  int data;
};
```

[12] Because a map can be used only in ghost code, the issue of runtime memory consumption does not apply to it.

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

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, then its data is present in the set val; this binds the values stored in the implementation to the abstract representation

You may note that the set val is under-specified: it might be that it has some elements not stored in the list. We'll get back to this issue later. Now let's have a look at the specification of a function adding a node to the list:

```
int add(struct List *l, int k)
  _(requires \wrapped(l))
  _(ensures \wrapped(l))
  _(ensures \result != 0 ==> l->val == \old(l->val))
  _(ensures \result == 0 ==>
        \forall int p; l->val[p] == (\old(l->val)[p] || p == k))
  _(writes l)
```

The writes-clause and contracts about the list being wrapped are similar to what we've seen before. Then, there are the contracts talking about the result value. This function might fail because there is not enough memory to allocate list node, in such case it will return a non-zero value (an error code perhaps), and the contracts guarantee that the set represented by the list will not be changed. However, if the function succeeds (and thus returns zero), the contract specifies that if we take an arbitrary integer p, then it is a member of the new abstract value if and only if it was already a member before or it is k.

In other words, the new value of l->val will be the union of the old abstract value and the element k. Ideally, this contract is all that the caller will need to know about that function: what kind of effect does it have on the abstract state. It doesn't specify if the node will be appended at the beginning, or in the middle of the list. It doesn't talk about possible duplicates or memory allocation. Everything about implementation is completely abstracted away. Still, we need a concrete implementation, and here it goes:

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

We allocate the node, unwrap the list, initialize the new node and wrap it (we want to wrap the list, and thus everything it is going to own will need to be wrapped beforehand; the list is wrapped at the end of the unwrapping block), and prepend the node at the beginning of the list. Then we update the owns set (we've also already seen that). Finally, we update the abstract value using a ***lambda expression***. The expression \lambda T x; E returns a map, which for any x returns the value of expression E, which can reference x. If the type of E is S, then the type of map, returned by the lambda expression, is S[T]. An assignment m = \lambda T x; E has a similar effect to the following assumption (note that E will most likely reference x):

```
_(assume \forall T x; m[x] == (E))
```

Unlike assumptions, lambda expressions do not compromise soundness of the verifier. Just like for assumptions, the expression is always evaluated in the state as it was when the lambda was first defined, for example:

```
int x = 1;
int m[int] = \lambda int y; y + x;
_(assert m[0] == 1) // succeeds
x = 2;
_(assert m[0] == 1) // still succeeds
```

One can imagine, that when this lambda expression is defined, VCC will iterate over all possible values of x, and store the value of E in m[x]. Lambda expressions are much like function values in functional languages or delegates in C#.

The body of our lambda expression shows similarity to the body of the quantifier we have used in specification. It doesn't, however, have to be the same:

```
int[int] foo(int v)
  _(ensures \forall int x; x >= 7 ==> \result[x] >= v)
{
  return \lambda int y; (y&1) == 0 ? INTMAX : v;
}
```

Thus, the specifications for lambda expressions can hide information.

7.1 Expressing reachability

The following subsection, till the beginning of § 8 (which is about concurrency), might be somewhat difficult upon first reading of this tutorial. It is not required to understand § 8.

At minimum the list should support adding elements and checking for membership. For example, we would expect:

```
int member(struct List *l, int k)
  _(requires \wrapped(l))
  _(ensures \result != 0 <==> l->val[k])
```

Our current list invariant is strong enough only to show \result != 0 ==> l->val[k], because it only says that if the list owns something, then it's in the val. It also says that if something can be reached by following the next field from the head, then it is owned. What we want to additionally say is that if something is in the val set, then it can be reached from the head. Unfortunately, such property is not directly expressible in first-order logic (which is the underlying logic of VCC specifications). To work around this problem we associate with each node the set of values stored in all the following nodes and the current node. Additionally we say that the set for NULL node is empty. This way, as we walk down the next pointers we can keep track of all the elements that can be still reached. Once we reach the NULL pointer, we know that nothing more can be reached. The set of reachable nodes are stored as maps from int to bool. We need one such map per each node, so we just put a ghost map from struct Node* to the sets. Alternatively, we could store these sets as a field inside of each node, but maps gives more flexibility in updating it using lambda expressions.

```
_(dynamic_owns) struct List {
  _(ghost bool val[int];)
  struct Node *head;
  _(ghost bool followers[struct Node *][int];)
  _(invariant val == followers[head])
  _(invariant head != NULL ==> \mine(head))
  _(invariant followers[NULL] == \lambda int k; \false)
  _(invariant \forall struct Node *n;
                  \mine(n) ==> n->next == NULL || \mine(n->next))
  _(invariant \forall struct Node *n;
                  \mine(n) ==>
                     \forall int e;
                        followers[n][e] <==>
                        followers[n->next][e] || e == n->data)
};
```

All these changes in the invariant do not affect the contract of add() function, and the only change in the body is that we need to replace the update of l−>val with the following:

```
      l->followers[n] =
        (\lambda int z; l->followers[n->next][z] || z == k);
      l->val = l->followers[n];
```

That is adding a node at the head only affect the followers set of the new head, and the followers sets of all the other nodes remain unchanged. Now let us have a look at the member() function:

```
int member(struct List *l, int k)
  _(requires \wrapped(l))
  _(ensures \result != 0 <==> l->val[k])/*{endspec}*/
{
  struct Node *n;

  for (n = l->head; n; n = n->next)
    _(invariant n != NULL ==> n \in l->\owns)
    _(invariant l->val[k] <==> l->followers[n][k])
  {
    if (n->data == k)
      return 1;
  }
  return 0;
}
```

The invariants of the for loop state that we only iterate over nodes owned by the list, and that at each iteration k is in the set of values represented by the list if and only if it is in the followers set of the current node. Both are trivially true for the head of the list, for the first iteration of the loop. For each next iteration, the invariant of the list tells us that by following the next pointer we stay in the owns set. It also tells us, that the followers[n−>next] differs from followers[n] only by n−>data. Thus, if n−>data is not val, then the element, if it's in followers[n] must be also in followers[n−>next].

7.2 Sorting revisited

In § 5.1 we have considered the bozo-sort algorithm. We have verified that the array after it returns is sorted. But we would also like to know that it's a permutation of the input array. To do that we will return a ghost map, which states the exact permutation that the sorting algorithm produced.

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

_(typedef unsigned perm_t[unsigned]; )

_(logic bool is_permutation(perm_t perm, unsigned len) =
  (\forall unsigned i, j;
     i < j && j < len ==> perm[i] != perm[j]))

_(logic bool is_permuted(\state s, int *buf, unsigned len,
                            perm_t perm) =
  \forall unsigned i; i < len ==>
      perm[i] < len && \at(s, buf[ perm[i] ]) == buf[i])

_(logic perm_t swap(perm_t p, unsigned i, unsigned j) =
      \lambda unsigned k; k == i ? p[j] : k == j ? p[i] : p[k])

void bozo_sort(int *buf, unsigned len _(out perm_t perm))
  _(writes \array_range(buf, len))
  _(ensures sorted(buf, len))
  _(ensures is_permutation(perm, len))
  _(ensures is_permuted(\old(\now()), buf, len, perm))
{
  _(ghost \state s0 = \now() )

  _(ghost perm = \lambda unsigned i; i)

  if (len == 0) return;

  for (;;)
    _(invariant \mutable_array(buf, len))
    _(invariant is_permutation(perm, len))
    _(invariant is_permuted(s0, buf, len, perm))
  {
    int tmp;
    unsigned i = random(len), j = random(len);

    tmp = buf[i];
    buf[i] = buf[j];
    buf[j] = tmp;
    _(ghost perm = swap(perm, i, j) )

    for (i = 0; i < len − 1; ++i)
      _(invariant sorted(buf, i + 1))
    {
      if (buf[i] > buf[i + 1]) break;
    }

    if (i == len − 1) break;
  }
}
```

This sample introduces two new features. The first is the output ghost parameter _(out perm_t perm). We use it when we need a function to return something in addition to what it normally returns. To call bozo_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_t myperm; )
  // ...
  bozo_sort(buf, len _(out myperm));
}
```

The value is only copied on exit of bozo_sort(), though during its execution it has its own copy. It is thus different than passing a pointer to the local. It is also more efficient for the verifier.

The second, somewhat more advanced, feature is explicit state manipulation. The function \now() returns the current state of the heap (i.e., dynamically allocated memory; in future it will also work for locals, but for now it only applied to memory location, address of which was taken). The state is encapsulated in a value of type \state. The expression \at(s, E) returns the value of expression E as evaluated in state s. You can see \old(...) as a special case of this.

Thus, the algorithm maintains the map containing the current permutation of the data, with respect to the initial data (we store the initial state in s0). The initial permutation is just the identity, and whenever we swap elements of the array, we also swap elements of the permutation.

Exercises

1. Write and verify an iterative program that sorts an array of ints using bubblesort. The specification should be the same as for bozo-sort above.

8. Atomics

Writing concurrent programs is generally considered to be harder than writing sequential programs. Similar opinions are held about verification. Surprisingly, in VCC the leap from verifying sequential programs to verifying fancy lock-free code is not that big. This is because the verification in VCC is inherently based on invariants: conditions that are attached to data and need to hold *no matter which thread* accesses it.

But let us move from words to actions, and verify a canonical example of a lock-free algorithm, which is the implementation of a spin-lock itself. The spin-lock data-structure is really simple – it contains just a single boolean field, meant to indicate whether the spin-lock is currently acquired. However, in VCC we would like to attach some formal meaning to this boolean. We do that through ownership – the spin-lock will protect some object, and will own it whenever it is not acquired. Thus, the following invariant should come as no surprise:

```
_(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 for the regular C compiler (i.e., it makes the compiler assume that the environment might write to that field, outside the knowledge of the compiler). For VCC it means that the field can be written also when the object is closed (that is after wrapping it). The idea is that we will not unwrap the object, but write it atomically, while preserving its invariant. 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; normally this is not possible).

First, let's have a look at lock initialization:

```
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)
  })
}
```

One new thing there is the use of ***ghost parameter***. The regular lock initialization function prototype does not say which object the lock is supposed to protect, but our 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)).

Second, we require that the object to be protected is wrapped (recall that wrapped means closed and owned by the current thread). We need it to be closed because we will want to make the lock own it, and lock can only own closed objects. We need the current thread to own it, because ownership transfer can only happen between the current thread and an object, and not for example some other thread and an object. Third, we say we're going to write the protected object. This allows for the transfer, and prevents the calling function from assuming that the object stays wrapped after the call. Note that this contract is much like the contract of the function adding an object to a container data-structure, like sc_add() from § 6.3.

Now we can see how we operate on volatile fields. We shall start with the function releasing the lock, as it is simpler, than the one acquiring it.

```
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 have a look at the contract. Release() requires the lock to be wrapped.[13] The preconditions on the protected object are very similar to the preconditions on the InitializeLock(). Note that the Release() does not mention the lock in its writes clause, this is because the write it performs is volatile. Intuitively, VCC needs to assume such writes can happen at any time, so one additional write from this function doesn't make a difference.

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, § 6.1), and we check the invariant at the end of the atomic block.

The question arises, what guarantees that other threads won't interfere with the atomic action? VCC allows only one physical memory operation inside of an atomic block, which is indeed atomic from the point of view of the hardware. Here, that 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. However, inside our atomic block we can also see the update of the owns set. This is fine, because the ghost code is not executed by the actual hardware.

The reason we can use ghost code is a simulation relation between two machines. Machine A executes the program with ghost code, and machine B executes the program without ghost code. Because ghost code cannot write physical data or influence the control flow of physical code in any way, the contents of physical memory of machines A and B is the same. Therefore any property we prove about physical memory of A also holds for B. Now, if we imagine that both machines are multi-threaded, and the machine A blocks other threads when it's executing ghost code, the same simulation property will still hold.

---

[13] You might wonder how multiple threads can all own the lock (to have it wrapped), we will fix that later.

It is not particularly difficult to see that this atomic operation preserves the invariant of the lock. But this isn't the only condition imposed by VCC here. 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, should we forget to mention l−>protected_obj in the writes clause VCC will complain about:

```
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.
```

As another example, should we forget to perform the ownership transfer inside of Release(), VCC will complain 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.
```

Let's then move to Acquire(). The specification is not very surprising: it requires the lock to be wrapped, and ensures that after the call the 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_add() function from § 6.3.

```
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 built-in, 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. One can define them by suppling a body. It is presented only to the VCC compiler (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;
  }
}
```

This is one of the places where one needs to be very careful, as there is no way for VCC to know if the definition you provided matches the semantics of your regular C compiler. Make sure to check with the regular C compiler manual for exact semantics of its built-in functions.

We plan to include a header file with VCC containing a handful of popular operations, so you can just rename them to fit your compiler.

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 § 6.5. 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 § 6.5, 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 (§ 6.1), 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.

A. Triggers

[**TODO:** The current VCC version needs to be told to infer triggers, see § **??**]

The triggers are likely the most difficult part of this tutorial. As of July 2010, the trigger inference algorithm in VCC (as opposed to the SMT solver) has been implemented, so triggers need to be used less often. Thus, we didn't need any trigger annotations for the examples in the tutorial. Still, you'll 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.

A.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)
```

A.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}.

A.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.

B. Memory model

In most situations in C the type of a pointer is statically known: while at the machine code level the pointer is passed around as a type-less word, at the C level, in places where it is used, we know its type. VCC memory model makes this explicit: pointers are understood as pairs of their type and address (an word or integer representing location in memory understood as an array of bytes). For any state of program execution, VCC maintains the set of ***proper pointers***. [**TODO:** we might want a better name] Only proper pointers can be accessed (read or written). There are rules on changing the proper pointer set — e.g., one can remove a pointer (T∗)a, and add pointers (char∗)a, (char∗)(a+1), . . . , (char∗)(a+sizeof(T)−1), or *vice versa*. These rules make sure that at any given time, representations of two unrelated proper pointers do not overlap, which greatly simplifies reasoning. Note that given a struct SafeString ∗p, when \proper(p) we will also expect \proper(&p−>len). That is, when a structure is proper, and thus safe to access, so should be all its fields. This is what "unrelated" means in the sentence above: the representations overlap if and only if they pointer refer to a struct and fields of that struct. It is OK that fields overlap with their containing struct, but that structs overlap each other.

B.1 Reinterpretation

C. Overflows and unchecked arithmetic

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 _(unchecked)E == E if E fits in the proper range, and some other basic facts about (unsigned)−1. If you need anything else, you will need to resort to bit-vector reasoning (§ D).

D. Bitvector reasoning

Remember our first min() example? Surprisingly it can get more involved. For example the one below does not use a branch.

```
#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)));
}
```

```
Verification of min succeeded.
Verification of min#bv_lemma#0 succeeded.
Verification of min#bv_lemma#1 succeeded.
```

The syntax:

```
_( assert {bv} \forall int x; (x & −1) == x )
```

VCC to prove the assertion using the fixed-length bit vector theory, a.k.a. machine integers. This is then used as a lemma to prove the postcondition.

E. Other VCC Features

This appendix provides short description of several other features of the VCC annotation language and the verifier itself.

E.1 Pure functions (main text?)

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. The function min() from § 3 is an 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.

E.2 BVD (main text)

E.3 addr-eq(), addr()

E.4 arrays-disjoint

E.5 begin-update

E.6 start-here

E.7 Using pure functions for triggering

E.8 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). The advantages of this separation 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.

Block contracts are not allowed if the block contains a return, or a goto to a label outside the block.

[**TODO:**] talk about _(requires \full_context())

E.9 Frame axioms

E.10 extent-mutable and friends

E.11 Claim upgrade

E.12 by-claim

E.13 Approvers

E.14 set operations

E.15 Volatile domains (?)

E.16 Globals

E.17 Groups

E.18 Structure inlining + backing member

E.19 Structure equality

E.20 Out parameters

E.21 Skinny expose

E.22 Mathint

E.23 Allocating ghost objects

E.24 Smoke

E.25 known (?)

E.26 Isabelle

F. Soundness

[**TODO:** Should we have a caveat list as an appendix to the tutorial?]

G. Solutions to Exercises

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.