VCC Syntax Refresh

Ernie Cohen, Michał Moskal, Wolfram Schulte, Stephan Tobies

This document describes our initial thoughts about updating the annotation language of VCC to make it simpler and more consistent. Our starting point is the existing ACSL specification (ANSI/ISO C Specification Language), which is used by the Frama-C project and is available from

<http://frama-c.cea.fr/download/acsl_1.4.pdf>

This document follows the structure of the ACSL spec and outlines the major differences and how to plan (if any) to resolve them with the ACSL standard.

### 1.2 Generalities

Instead of using comments to hide the annotations from the normal C compiler, we will use a macro:

#ifdef \_\_VCC\_DONT\_DEFINE\_UNDERSCORE  
# define \_ \_\_vcc\_spec  
#endif  
#ifndef \_\_VCC\_VERIFY  
# define \_\_vcc\_spec(...) /\* nothing \*/  
#endif

Thus, unless the user decides to use some other identifier, all the specifications will be inside \_(...).

This allows preprocessing to work as usual, in particular macros like:

#define READER\_COUNT(n) ((n) >> 1)

can be freely used in the specifications.

Multi-line annotations are allowed and need to be separated by semicolon:

\_( requires x < 5;  
 ensures y > 10 )

### 2.1 Lexical rules

As ACSL, we will support the use of a number of Unicode chars in annotations (e.g. ≤, ∀, ∧, ⇒); we might want to consider VS editor support to replace \forall etc. with these special forms using an ‘autocorrect’ approach.

Identifiers can start with the backslash character and any annotation-specific keyword does so (see ‘\forall’ below), except when the keyword is at the start of a spec block (like the ‘requires’ in the example above) or after a semicolon that starts new annotation (like the ‘ensures’ above).

### 2.2 Logic expressions

* Unlike ACSL, we do **not** distinguish between terms and predicates. We follow the ACSL syntax for quantification:

\forall int x,y, uint z; foo(x,y,z)

* We allow triggers after the ‘;’ as in \forall obj x; {bar(x)} foo(bar(x))
* We add the support for \let , for example \let x = f(y, 12); g(x, x)
* For functional modifiers of struct or maps, we add the syntax { s \with .id = v }, this replaces our current syntax s / { .id = v }. ACSL uses for instead of \with, we will also support that.
* We do not add the ACSL’s bitwise Boolean operators ‘-->’ or ‘<-->’.
* Unlike ACSL, we do not support consecutive comparison operators of the form a <= b < c because the C semantics of that expression is confusingly different. It also makes moving code between implementation and spec harder.

### 2.2.4 Integer arithmetic and machine integers

ACSL diverges significantly from VCC in its use of mathematical integers. In ACSL specification, all machine integers types are implicitly converted to the mathematical integer type ‘integer’ (‘mathint’ in VCC, which we will rename). Conversion back to machine integer types must be explicit and will always succeed by taking the “unique value of the corresponding type that is congruent to the mathematical result modulo the cardinal of this type, that is ”

We are currently considering this approach, too. It will solve a number of semantic problems that we are encountering, with the following additions:

* Using the notation \C(e), the expression e will be evaluated using (unchecked) C semantics.
* We want to have a way to express checked semantics that assert absence of overflows in implementation code. How?

We would be really interested in getting feedback on this point!

### 2.2.5 Real numbers and floating point numbers

For now, no support beyond what we currently have (i.e., very limited).

### 2.2.6 Structures, unions, arrays in logic

ACSL uses a separate syntax for logic type definitions. C already has syntax for typedefs and we would use that instead of a second mechanism for defining types, i.e.:

\_(ghost typedef int intmap[int])

We will introduce a separate \record structured type, which replaces our current kludge of ‘struct vcc(record) { … }’

### 2.2.8 Address operator

As our ownership system relies on pointers also for spec objects, we do support taking of addresses of spec objects and stick with our current distinction between implementation pointers (T\*) and specification pointers (T^).

#### 2.3 Function Contracts

* Function contracts in VCC will follow the function prototype, not precede it, as done in ACSL.
  + Motivation: Our \_(...) annotations look more first class than ACSL comments, so it would make sense to have definitions of formal parameters **before** their usage in contracts.
* The ACSL annotations for termination are orthogonal to our current efforts and can optionally be added in later. If so, support general termination measures, not only integers.
* Named behaviors have low priority, but if we should adopt them, we would follow ACSL’s example
* In ACSL, no writes clause means unspecified with a provision for tool inference should the function be defined. For VCC, no writes clause means no writes, and we tend to like to keep it that way.
* ACSL allows merging of multiple contracts. We insist on having all contracts in one place, modulo a support for public/private contracts, which have been requested for VCC, but for which we have no proper answer yet.
* For framing, we would diverge from ACSL and write ‘writes p’ where they would write ‘assigns \*p’; we think that the pointer-oriented approach is more suitable for our object-oriented memory model (writes p means writing the ownership domain of p, not the structure pointed to by p). We would use ‘writes’ instead of ‘assigns’ to disambiguate our approach from theirs.
* We will adopt the following syntax for arrays:
  + as\_array(p, 10)will be replaced by (int[10])p
  + array\_range(p, 10) will be replaced by p + (0 .. 9) or, equivalently, &p[0 .. 9] I am still not 100% happy with this second thing.

### 2.4 Statement annotations

* Asserts become their own annotation elements: \_( assert p ) instead of assert(p) so we do not collide with assert.h; if desired, the assert function from assert.h can get a precondition that requires the condition to hold.
* We will add an \_( assume ... ) statement (absent from ACSL).
* Similar to function contracts, loop contracts should follow the loop head and not precede it.
* It seems that we already implement the same notion of loop invariant as ACSL specifies, no semantic difference there.
* We are unclear about the use of invariants in the middle of a loop body – we will need to check how this would translate to Boogie.
* Their general invariants also would provide support for loops created by gotos – we should consider adding support for that, too (which we used to have in VCC1).
* ACSL’s approach of using \at with labels seems questionable: what if the label had not been passed on the way to the current state, or multiple times? Instead, we stick with our support of states as first-class citizens. We will need clean up our story regarding the use of old for locals and non-head parameters; also, the different semantics of \old in loops needs to be re-considered, I think.

### 2.6 Logical specification

We would slightly change our current way of specifying pure functions to:

spec(   
 pure bool isPositive(int x)  
 ensures \result == x > 0;  
)

Note that pure is the specification keyword.

For the moment, such pure functions must not have a body. In the future, we may also consider allowing pure functions with bodies, which then provide a witness, and thus lifting the restriction that pure functions need to have an ensures clause of the form \result == .... For compatibility with ACSL, we could consider adding their syntax, too:

spec( logic bool isPositive(int x) = x > 0; )  
spec( predicate isPositive(int x) = x > 0; )

We would prefer ours because we do not need an entirely new class of declarations.

We also want to support spec functions with bodies (as we currently do), which would be indicated by the use of the keyword ‘procedure’ as in:

\_(  
 ghost bool update(obj o)  
 { … }  
)

Instead of ACSL’s spec( lemma foo: \forall ... ) we would rather have something like:

spec( procedure bool foo()  
 ensures \forall ...   
{ // possible to put some guidance to the prover here  
} )

We will have spec( axiom \forall ... ).

We also need a way to achieve what we currently do with bv\_lemma. Abstractly, this is nothing but a hint how certain assertions should be proven before they can be used, which would be reflected in the following sytax:

{  
 …  
 spec( assert {bv} \forall int i; … )  
}

Here, the {bv} serves as a pragma to VCC that suggest how the assert should be proved. One could think of other pragmas for similar purposes, thus we support them in this more general fashion.

* VCC will not support axiomatic definitions and recursive logic functions at this time. For now one can use axioms.
* \lambda will follow ACSL syntax.
* Higher-order logic constructions (\min, \max, \sum, ...): we want to support them, but Ernie doesn’t want them built in.
* Concrete logic types: datatypes (sum types) are unsupported at this time, records are already supported (with syntax the same as for defining structs, except with the \record keyword).
* Hybrid functions: currently we have objects representing state as first class, so one can just pass them to a spec function.
* Reads clause: we should double check if frame axioms are really used.
* Specification modules are unsupported at this time.

### 2.7 Pointers and physical addressing

\valid and \null will be supported. Equivalents of the \base\_addr, \block\_length and \offset are left to be defined by the implementor of the memory allocator.

\separated is largely subsumed by ownership, we do not support it at this time.

\fresh will be supported, \freed(s) amounts to specifying s in the writes clause and not talking about it in precondition, we do not support it.

### 2.8 Sets as first-class values

We keep supporting the old VCC ptrset type, we do not introduce polymorphic sets, one should use maps. We will support \union, \difference, \inter, \empty, and { a, b, c } syntax.

### 2.9 Abrupt termination

We do not support it at this time.

### 2.10 Dependencies information

We do not support it at this time.

### 2.11 Data invariants

We stick to our approach of defining invariants inside of types as this is central to our methodology.

### 2.12 Ghost variables and statements

We follow ACSL:

int y;  
\_(ghost int x);  
y = 7;  
\_(ghost x = 10);

struct Foo {  
 int xy;  
 \_(ghost int x)  
}

Volatile variables, in the sense of ACSL, are unsupported at this time.

### 2.13 Undefined values, dangling pointers

\initialized and \specified are unsupported (their usage is unclear to us).