Can C++ Be Made as Safe as SPARK?

David Crocker
Escher Technologies Ltd.
Mallard House, Hillside Road
Ash Vale, Aldershot GU12 5BJ, UK
+44 20 8144 3265
dcrocker@eschertech.com

ABSTRACT
SPARK offers a way to develop formally-verified software in a language (Ada) that is designed with safety in mind and is further restricted by the SPARK language subset. However, much critical embedded software is developed in C or C++. We look at whether and how benefits similar to those offered by the SPARK language subset and associated tools can be brought to a C++ development environment.

Categories and Subject Descriptors
F.3.1 [**Logics and meanings of programs**]: Specifying and Verifying and Reasoning about Programs – *assertions, invariants, mechanical verification, pre- and post-conditions, specification techniques.*

General Terms
Reliability, Security, Languages, Verification

Keywords
Formal methods; software verification; design by contract; C++; high integrity software

1. INTRODUCTION
The Ada language is generally considered to be well-designed from a safety perspective. The SPARK [1] subset, designed for use in developing high-integrity software, restricts the Ada language by removing constructs that are considered unsafe or difficult to reason about, and adds notation for information flow, function contracts, and other formal specifications. The SPARK tool set allows the annotated program to be analyzed, in particular it can generate verification conditions and attempt to prove them. The most recent versions of the SPARK tools use the function contract notation of Ada 2012 in preference to the older notation involving comments that start with a special character.

Despite the advantages of the Ada programming language and the SPARK subset, much critical embedded software is today written in C and there is an increasing use of C++ in this field. C and C++ are both supported on a wide range of embedded processors by a number of compilers and development environments of good quality. However, in comparison with Ada, from a correctness and safety perspective C is a poorly designed language. To mitigate this, subsets of C are widely used in developing high-integrity software, of which the best known is MISRA-C, now in its third version [2]. Many commercial static checking tools for C are available, and most of these can be configured to enforce MISRA-C compliance to a greater or lesser extent.

Unfortunately, C also lacks important features, such as encapsulation at the object level, and generics. Lack of object encapsulation makes it difficult to protect data from unintended modification and enforce object invariants, except where there is only a single instance of the type. Lack of generics means that either the same code is written more than once to work on different types, or type safety has to be sacrificed by using void pointers for parameter passing, for example in the standard library functions *qsort* and *bsearch*. C++ provides object-level encapsulation by supporting classes, and generics by supporting template declarations. So when developing critical software, a subset of C++ could offer advantages over MISRA-C, given suitable support from static checking and verification tools.

There are already subsets of C++ designed for high-integrity software, notably MISRA-C++ 2008 [3] and JSF-C++ [4]. Both are based on sets of rules that prohibit the use of various features of C++. We have concerns over the use of these subsets in the most critical software, for example at Safety Integrity Level 4 which is where formal verification is most often performed. Our concerns are these:

- C++ is a large and complicated language. It is difficult to identify all the weaknesses in it that may contribute towards inadvertent errors.
- Because of its size, it is difficult to develop formal semantics and associated verification tool support for the language in its entirety.

In practice, critical embedded software developers do not need the full C++ language. Dynamic memory allocation is generally prohibited in critical embedded software, and this in turn limits the C++ features that can be used.

We therefore propose a different approach. Most well-written C programs are also valid C++ programs, and programs that are valid in both C and C++ have the same semantics in both languages, provided that a small number of potential issues are avoided. So our approach is:

- Start with the MISRA-C:2012 subset of C;
- Add selected features from C++ along with rules limiting how they may be used;

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
HILT 2014, October 18-21, 2014, Portland, Oregon, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3217-0/14/10…$15.00
http://dx.doi.org/10.1145/2663171.2663179

- Further constrain the MISRA-C:2012 rules by banning C constructs that are not needed because better C++ constructs can replace them, and by banning constructs that can have different semantics in C and C++;
- Add notation for adding SPARK-like function contracts and other specifications to C++ source code;
- Implement formal verification of all constructs permitted by the resulting C++ subset, using the Verified Design-by-Contract paradigm [20].

Our aim was to produce a C++ language subset that is sufficient for critical embedded software development, that addresses the main safety-related limitations of C, that is amenable to formal verification, and for which we can provide tool support in a reasonable time frame.

In keeping with the theme of this conference, this paper addresses the contribution to software safety that is made by the programming language and the analysis techniques that it supports. There are of course many other factors that contribute to software safety.

2. DESIGN OF THE C++ LANGUAGE SUBSET

2.1 C++ Safety Issues Inherited From C

The C++ language [5, 6] includes several features that do not fit well with safety. Fortunately for us, most of them are inherited from C, and the C language has been widely-studied in relation to safety. Some known vulnerabilities in a number of programming languages, including Ada, SPARK and C, are listed in [7]. Unfortunately, that publication does not list vulnerabilities in C++. Vulnerabilities in C are also listed in the MISRA-C:2012 guidelines [2], along with rules to avoid them. Examples of such vulnerabilities include: excessive automatic type conversion, identifier re-use, accidental use of '=' where '==' was intended, and operator precedence issues. When the MISRA-C guidelines are enforced, these vulnerabilities are for the most part avoided. Enforcing some of them requires the use of formal techniques in the general case. Almost all the MISRA-C 2012 guidelines can be mapped directly or with only minor modifications to our subset of C++.

Many vulnerabilities of C fall into the categories of undefined, unspecified and implementation-dependent behaviour. For the most part, we can avoid these behaviours by defining appropriate preconditions for the constructs concerned, and using formal verification to ensure that these preconditions are satisfied. This is akin to proving exception freedom in SPARK. Some implementation-defined behaviours cannot be avoided, and for those we provide tool configuration options to describe the behaviour of the compiler and platform. For example, we provide a means to specify the ranges of the built-in integral types, and whether integer division of a negative by a positive number rounds up or down.

There are some particular weaknesses in C that cannot be mitigated simply by subsetting the language. Three of them are concerned with pointers. Whereas SPARK bans the use of Ada access types, in C it is necessary to use pointers at least for passing array parameters between functions. However, the C language does not distinguish between a pointer to an object and a pointer to an array of objects. On receiving a pointer to an array, it is not possible to inquire of the pointer how many elements are in the array. Furthermore, C allows any pointer type to take the value NULL; but in most instances of using pointers to pass parameters, a null pointer is not an appropriate value.

We have already solved two of these problems in our earlier tool [8]. When a pointer is declared using standard C syntax, our tool requires that every initialization of or assignment to that pointer is not the value NULL. Furthermore, use of the array index and pointer arithmetic operators on such a pointer is forbidden. To specify a pointer to an array of objects, the user adds the keyword **array**. Such a pointer may only be initialized or assigned to point to an array, not to a single object. To specify that NULL is an allowed value, the user adds the keyword **null**. For example:

```
char * p1;            // pointer to a single
                      // character
char * array p2;      // pointer to an array
                      // of characters
char * null p3;       // pointer to a single
                      // character, or NULL
char * array null p4; // pointer to an array
                      // of characters or
NULL
```

In order that the code may still be compiled by a standard C or C++ compiler, we define C preprocessor macros that cause **array** and **null** to be expanded to nothing, except when the source code is being analyzed by our tool. These macros are defined in a header file that is referenced at the start of each C source file by a **#include** directive. We also provide a **not_null**(...) macro for asserting that a nullable pointer is not null and converting its type to the corresponding non-nullable pointer.

Although C++ does not include bounds information in pointers to arrays, for verification purposes we pretend that it does. We do this by augmenting the array pointer type with 'ghost' fields. A ghost entity is one that can be used in a specification construct, but not in executable code. We provide a ghost field to yield the number of elements in the array, and another to yield the offset of the array pointer into the array (because a C++ array pointer can point part way into an array). These ghost fields can be referred to in function contracts and other formal specifications, allowing us (for example) to write preconditions to ensure that accesses to array elements are within the bounds of the array.

Where the bounds of an array that is passed as a function parameter need to be available not just in the specification but in the code as well, we adopt the notion of a generic Array class that was recommended by the JSF-C++ standard. Unfortunately, the *array.doc* file describing the JSF-C++ Array class has not been put in the public domain. We therefore defined our own *array_ref<T>* class template for passing array parameters. This class is small enough to be passed by value, and it contains both the pointer to the start of the array and the number of array elements. It can also be configured to perform run-time bounds checks when indexing into the array, if this is desired.

Another weakness of C is that there is a single type conversion syntax that is used not only for expressing relatively safe conversions (for example, converting between different integral

types) but also for more dangerous conversions, such as converting between pointer types or casting away **const**. We avoid this issue by mandating use of the C++ type conversion operators for these more dangerous conversions, as described in the next section.

2.2 C++ Constructs Included in our Subset

Our C++ subset is focused on including those features of C++ that offer significant benefits over C to the writers of high-integrity embedded software, while leaving out features that are of doubtful utility or safety, or for which we feel that the safety implications are not well understood.

The first items in our list of C++ constructs to include are the C++ type conversion operators **static_cast**, **reinterpret_cast** and **const_cast**. The C++ conversion operators show what sort of conversion is intended, and are therefore safer to use than the single type casting notation of C. Therefore, we mandate the use of C++ type conversion notation for the more dangerous forms of conversion. We continue to allow the C type casting operator to be used where it has the same meaning as a **static_cast** to the same type.

Next on our list are classes, in order to provide the object-level encapsulation that is missing from C. A further benefit of using classes is that if a class has at least one declared constructor, then it is impossible to declare or create an instance of that class without a constructor being called. Our subset requires all classes to have at least one constructor, and all constructors to initialize all member variables of the class. In this way, use of uninitialized or partially-initialized objects is avoided.

Use of class inheritance and dynamic binding remains controversial in high-integrity software. However, there is increasing interest in using these techniques in some sectors, notably aviation. When doing formal verification of source code annotated with function contracts, it is relatively straightforward to ensure that local type consistency is satisfied as required by the DO-332 Object Oriented Supplement to DO-178C [9]. Furthermore, we consider that use of inheritance and dynamic binding is preferable to using function pointers, which is the usual solution adopted by C programmers faced with similar requirements. We therefore include single inheritance and virtual functions in our subset.

Support for function templates is needed in order to allow generic functions to be written without sacrificing strong typing, as discussed earlier. Class templates are needed, both to support the *array_ref<T>* class and to support other useful classes, such as a bounded vector class. It is less obvious that template specialization is needed and safe to use, so for the time being we have not included it in our subset.

Our C++ subset also permits side-effect free user-defined operator declarations (but not type conversion operators), and reference types. These are needed to support the *array_ref<T>* class as well as being useful features in their own right.

2.3 Mitigating Unsafe Features of C++

Although C++ inherits a lot of poorly-designed language features from C, most of the new constructs it adds to C are fairly well-designed, in our opinion. C++ strengthens the type system of C a little, despite the issues with migrating C code to C++ that this causes. However, C++ introduces a small number of unsafe new features that require mitigation for high-integrity software.

One such feature is the treatment of string literals. Although a string literal yields type **const char*** in most C++ contexts, in some contexts it can be implicitly converted to **char***. This was done to improve backwards-compatibility with C. We ban the use of this conversion in our subset.

Another concerns the overloading of functions, operators and constructors by argument number and type. We are forced to support overloading in our C++ subset, because it is needed to support class constructors and user-defined operators. Unfortunately, overloading interacts badly with implicit type conversion of actual parameters. A call to a function, operator or constructor may potentially match more than one of the overloaded declarations, depending on which implicit type conversions are applied to the actual parameters. The C++ rules for resolving such ambiguities are complex and occasionally give surprising results. Therefore, our subset requires that if a call potentially matches more than one overloaded declaration, then it must match one of them without the use of automatic type conversions.

When a class constructor is declared with a single parameter, that constructor introduces an implicit type conversion from the argument type to the class type, unless the constructor is flagged **explicit**. So in common with MISRA-C++ and JSF-C++, we require all single-argument constructors to be declared **explicit**.

When a derived class declares a function with the same name and parameter types as a virtual function in the parent class, it overrides the parent class function. Such overriding could be inadvertent. The 2011 revision of the C++ language standard [6] provides a way of indicating intentional overriding by adding the reserved identifier **override**. We elevate **override** to the status of a keyword and mandate its use. When using a C++ compiler that implements the older 2003 C++ standard, to preserve compatibility we define **override** as a macro expanding to nothing in the usual way.

Other safety-enhancing language additions in C++ 2011 include the **final** reserved identifier and the **nullptr** keyword. Again, we allow these in our subset and we define macros to make them acceptable to older C++ compilers.

When a C++ program declares statically-allocated variables that need to be constructed, the order in which these initializations are performed is defined for the declarations within a single translation unit, but not between different translation units. This raises the possibility that the initialization of a statically-allocated object in one translation unit could depend on the value of another statically-allocated object in another translation unit that has not yet been initialized. Our subset therefore prohibits the declaration of any statically-initialized object whose initialization is non-trivial and whose value depends on the value of an object with non-trivial initialization in another translation unit. In contrast, the Ada language requires the translator to determine a suitable elaboration order for all the packages that make up the program.

2.4 Expressing Contracts and Other Specifications

Unlike Ada 2012, C++ does not have built-in language constructs for expressing function contracts or other

specifications, apart from a macro for declaring assertions. Fortunately, such constructs can be readily added to C++ by choosing suitable keywords such as **pre** and **post**, following the keyword by a bracketed list of expressions, and once again defining these in a header file as macros that expand to nothing. The C++ preprocessor discards the entire specification construct when the source file is processed by a standard C++ compiler. We prefer this approach over the alternative custom of expressing specifications as specially-formatted comments, because it gives specifications the visual impact associated with source code rather than comments, and text editors that understand C++ will perform their usual syntax highlighting on the expressions in the specifications. Many text editors for C++ can also be configured to treat **pre**, **post** etc. as additional keywords and highlight them appropriately.

A minor annoyance is that macros in C++ 2003 must have a fixed number of parameters, so comma cannot be used as a separator in specification expression lists. We use semicolon instead.

The main specification constructs we support are listed in Table 1. Where a construct supports a list of Boolean expressions, the individual expressions are conjoined implicitly by 'and', although separate verification conditions are generated for each expression.

Table 1. Primary specification constructs

pre(*expression-list*)	Declares preconditions
post(*expression-list*)	Declares postconditions
returns(*expression*)	Declares the value returned by a function. Equivalent to **post**(**result** == *expression*) except that recursion is permitted in *expression*
assert(*expression-list*)	Asserts conditions
invariant(*expression-list*)	Used in class declarations to declare class invariants, and in **typedef** declarations to declare constraints
keep(*expression-list*)	Declares loop invariants
decrease(*expression-list*)	Declares loop variant or recursion variant expressions
writes(*lvalue-expression-list*)	Declares what non-local variables the function modifies. If a function is declared without a writes-clause, then a default writes-clause is constructed based on the signature of the function.
assume(*expression-list*)	Declares predicates to be assumed without proof
ghost(*declaration-list*)	Declares ghost variables, functions, parameters etc.

2.5 Specification Expressions

Within specification macros such as **pre**(...) we allow side-effect-free C++ expressions. Of course this is not sufficient, and we add syntax for additional forms of expression. The main ones are listed in Table 2.

We add ghost members to a number of standard C++ types. For the array pointer types, the fundamental ones are *offset* (which returns the offset of the pointer from start of the array that it points into), *lim* (which returns the number of elements that can be addressed in a non-negative direction), and *all* (which yields the entire array addressed by the array pointer). For convenience we also provide *lwb* (lower bound i.e. lowest valid index, equal to -*offset*) and *upb* (upper bound, equal to *lim-1*), *isndec* (is-non-decrementing according to the < operator for the type) and *isninc*.

Table 2. Additional specification expressions

exists *identifier* **in** *expression* :- *predicate*	Existential quantification over the elements of *expression*, which must be an array or an abstract collection type
exists *type identifier* :- *predicate*	Existential quantification over all values of *type*
forall *identifier* **in** *expression* :- *predicate*	Universal quantification over the elements of *expression*, which must be an array or an abstract collection type
forall *type identifier* :- *predicate*	Universal quantification over all values of *type*
for *identifier* **in** *expression1* **yield** *expression2*	Applies the mapping function *expression2* to each element of collection *expression1*, yielding a new collection
those *identifier* **in** *expression1* :- *predicate*	Selects those elements of collection *expression1* for which *predicate* is true
that *identifier* **in** *expression1* :- *predicate*	Selects the single element of the collection *expression1* for which *predicate* is true
expression1 **in** *expression2*	Shorthand for **exists** *id* **in** *expression2* :- *id* == *expression1*, where *id* is a new identifier
expression **holds** *member*	*expression* must have union type, and *member* must be a member of that type. Yields true if and only if the value of *expression* was defined by assignment or initialization through *member*.
disjoint(*lvalue-expression-list*)	Yields true if and only if no two objects in the expression list have overlapping storage. Typically used in preconditions to state that parameters passed by pointer or reference refer to distinct objects.
operator **over** *expression*	Left-fold *operator* over collection *expression*. Used to express e.g. summation of the elements of an array.
old(*expression*)	When used in a postcondition, this refers to the value of *expression* when the function was entered. When used in a loop invariant, it refers to the value of *expression* just before the first iteration of the loop.

Recursion is usually prohibited by safety-critical software standards, however it is sometimes useful to write recursive specifications. In particular, it is useful to be able to specify the return value of a function recursively, and it is useful to be able to write a loop invariant that calls the containing function. We therefore allow recursion in these two specification contexts and forbid it elsewhere. The recursion is constrained to be finite in the usual way by means of a variant expression.

We support the use of C union types to implement variant records, but not for conversion between different types. We give each variable of union type a ghost discriminant, which records the name of the union type member through which it was last initialized or assigned. The **holds** expression allows this ghost discriminant to be queried.

```
#include "ecv.h"                          // for annotation macros
#include "stddef.h"                       // for size_t

const size_t capacity = 64;

template<class T> class Queue final
{
  T ring[capacity + 1u];                  // storage for the data
  typedef size_t invariant(value <= capacity) RingPointer;
                                          // range-limited type for head and tail pointers
  RingPointer hd, tl;                     // indices of the first and last elements of the buffer

public:
  ghost(
  _ecv_seq<T> abstractData() const        // retrieve function for data in the queue
    returns(
        (tl >= hd) ? ring.take(tl).drop(hd)
                   : ring.drop(hd).concat(ring.take(tl))
   );
  )

  bool empty() const                      // test if the queue is empty
  returns(abstractData().lim == 0)
  {
    return hd == tl;
  }

  bool full() const                       // test if the queue is full
  returns(abstractData().lim == capacity)
  {
    return (tl + 1u) % (capacity + 1u) == hd;
  }

  void add(T x) writes(*this)             // add an element to the queue
  pre(!full())
  post(abstractData() == old(abstractData()).append(x))
  {
    ring[tl] = x;
    tl = (tl + 1u) % (capacity + 1u);
  }

  T remove() writes(*this)                // remove an element from the queue
  pre(!empty())
  returns(abstractData().head())
  post(abstractData() == old(abstractData()).tail())
  {
    T temp = ring[hd];
    hd = (hd + 1u) % (capacity + 1u);
    return temp;
  }

  explicit Queue(T initVal)               // constructor to build an empty queue
  post(empty())
  {
    // Dummy initialization to satisfy the complete-initialization rule
    for (size_t i = 0; i <= capacity; ++i)
    writes(i; ring)
    keep(i <= capacity + 1)
    keep(forall k in 0..(i - 1) :- ring[k] == initVal)
    decrease(capacity + 1 - i)
    {
      ring[i] = initVal;
    }
    hd = 0;
    tl = 0;
  }
};
```

Listing 1: Annotated C++ code example

A complication is that standard first-order logic requires that all functions are total, which is not the case in programming languages. One solution to this is to use three-valued logic and write a theorem prover to work with it. The more common solution (which we adopt) is to use classical two-valued logic and to prove separately that the preconditions of all calls to partial functions are met. Even this is not sufficient, because some transformations to quantified expressions that are valid in classical first-order predicate calculus are no longer valid if the quantified expression includes calls to partial functions, even when the precondition of the expression as a whole is satisfied.

Our reasons for using a theorem prover rather than a constraint solver are largely historical. A constraint solver has the advantage that when a verification condition cannot be shown to be true, the constraint solver will often generate a counter-example. This can be translated back to the programming language to help the user understand the problem. On the other hand, from failed proof attempts our prover is sometimes able to infer additional preconditions and loop invariants that would make a proof possible, and we feed these back to the user as suggestions.

For the example in Listing 1 our tool generates 68 verification conditions and proves them all in less than three seconds. The proofs are saved to file so that they could, in principle, be checked by an independent tool.

3. FUTURE WORK

3.1 Namespaces

Although our present subset of C++ is sufficient to overcome the main limitations of C, we plan to add a few more C++ features. For example, C++ supports namespaces to help partition large projects and to avoid clashes between identifiers with external linkage declared by different modules. Namespaces are popular with C++ developers and are used in the C++ standard library. We intend to add namespaces to our C++ subset, subject to the results of an analysis of the safety implications of the argument-dependent name lookup, which is triggered when namespaces are used.

3.2 Template Instantiation Preconditions

It is often the case that C++ templates can often only be meaningfully instantiated when the types with which they are instantiated have applicable operators with certain semantics. For example, a sorting function may require the type to implement **operator<** in a way that defines at least a partial ordering. Early drafts of C++ 0x included a proposal [11] to declare these instantiation preconditions in the form of "template concept" declarations, but this feature did not make it into the C++ 2011 standard. Therefore we intend to add our own syntax to express them. Not only do instantiation preconditions guard against incorrect instantiation of class and function templates, they are also essential to allow such templates to be verified using formal techniques.

3.3 Semantics of Volatile Variables

C and C++ both support the **volatile** qualifier in variable declarations. These are variables whose values may change in ways that are not predictable by the compiler, and for which the order in which they are read and written must be strictly preserved. Our verifier implements the semantics of **volatile** variables as defined in the C++ language standard. However, when we attempted to perform formal verification on one particular embedded controller program, it became apparent that different sorts of volatile variables require different semantic treatment. For example, one use of **volatile** is to flag variables that are modified by an interrupt service routine and read by the main program thread. In the main program thread, these variables change unpredictably. Within the interrupt service routine, they do not, and it is desirable to treat them as normal variables so that their values can meaningfully be used within specifications. Another use of **volatile** is for variables that represent output ports. The value stored in the output port may be entirely predictable (unlike an input port) and it would be useful to refer to it in specifications; but the variable must still be declared **volatile** because the order in which it and other I/O ports are accessed must be preserved.

To resolve these issues, we plan to introduce some additional keyword macros, each of which expands to **volatile** but means something slightly different to our verifier. So we would be able to declare a variable that represents an output port whose value only changes when the program assigns it, and can therefore be used in specification expressions. Likewise, we would be able to declare a variable that is used to communicate between an interrupt service routine and the rest of the program. For such a variable, we would also provide a means to declare that it is effectively not volatile within a particular block of code, such as a block in which interrupts are disabled, or the interrupt service routine itself.

SPARK 2014 addresses a similar issue via the concept of external state [21].

3.4 Concurrency

Until recently, one of the biggest limitations of C and C++ was its lack of a standard for concurrency. The 2011 C++ language standard finally introduced concurrency primitives. The concurrency model has been formalized by Batty et al [12].

Shared-variable concurrency remains a big challenge for formal verification. We are delighted to be supporting the Taming Concurrency research project [22].

We have not yet attempted to model concurrency in our verification tool. This has not been a serious limitation so far. Developers generally apply formal verification only where the software has to meet the most demanding safety standards such as IEC61508 SIL 4, and such software typically runs single-threaded.

3.5 Floating Point Arithmetic

Target platforms for C++ and the associated libraries generally adopt the IEEE-754 floating-point standard [13]. From the perspective of formal verification, this standard defines the treatment of NaN (not-a-number) values in a very unhelpful way. In particular, a NaN is not equal to itself; so equational reasoning breaks down.

Our solution is to avoid allowing the program to generate NaNs. Our C++ subset forbids the use of the standard library functions that are provided to supply NaNs, and we put preconditions on functions such as *sqrt()* and inverse trigonometric functions so as to forbid input values that give rise to NaN outputs.

Another issue with floating point arithmetic is its inexact nature, which is at variance with the real arithmetic generally assumed by theorem provers. Our tool is currently unsound if floating-point arithmetic is used. For example, the theorem prover might

consider that if x can be proved to be nonzero, then x * (1.0 / x) == 1.0, but this is not true at execution time when x = 3.0. It would be possible to model IEEE arithmetic more accurately in the theorem prover by removing axioms for real arithmetic that are not valid for practical floating point arithmetic, but this greatly reduces the ability to prove useful things about the program. Range arithmetic may provide a partial solution to this issue.

4. RELATED WORK

The Larch/C++ project [14] defines an alternative annotation language for C++ modules. It is supported by tools for syntax and type checking, but does not appear to have been used to perform formal verification of critical C++ programs.

Several tools are available for performing formal verification of annotated C programs. These include the Jessie plugin [15] for Frama-C, Vcc [16], our own Escher C Verifier [17], and VeriFast [18]. The Vcc tool supports multithreaded C programs with variables shared between threads. Some of these tools have been compared by Rainer-Harbach [19].

5. CONCLUSION

As a starting point for a programming language subset for developing high-integrity software, C++ is less suitable than Ada 2012 because of its language design deficiencies and lack of function contracts. Nevertheless, we have demonstrated that it is possible to define a subset of C++ that is based on the MISRA-C 2012 subset of C while at the same time enhancing safety by including sufficient features of C++ to provide object encapsulation, generics, and other useful facilities. Along with this we have designed an annotation language to express function contracts and other specifications, and implemented a tool to generate and prove the associated verification conditions.

We therefore believe that where political or other considerations force the use of C++ rather than Ada, this choice of programming language need not of itself compromise safety, at least in single-threaded programs.

6. REFERENCES

[1] Barnes, John. “SPARK - The Proven Approach to High Integrity Software”. ISBN 978-0-957290-50-1, 2012.

[2] MIRA . “Guidelines for the Use of the C Language in Critical Systems”, ISBN 978-1-906400-10-1 (paperback), ISBN 978-1-906400-11-8 (PDF), 2013.

[3] MIRA. “Guidelines for the Use of the C++ Language in Critical Systems”, ISBN 978-906400-03-3 (paperback), ISBN 978-906400-04-0 (PDF), 2008.

[4] Lockheed Martin. “Joint Strike Fighter Air Vehicle C++ Coding Standards for the System Development and Demonstration Program”, Document Number 2RDU00001 Rev C, December 2005.

[5] ISO/IEC 14882:2003, “Programming languages – C++”, 2003.

[6] ISO/IEC 14882:2011, “Programming languages – C++”, 2011.

[7] ISO/IEC TR 24772:2013 “Guidance to avoiding vulnerabilities in programming languages through language selection and use”, second edition.

[8] Crocker, David, and Judith Carlton. "Verification of C programs using automated reasoning." Software Engineering and Formal Methods, 2007. SEFM 2007. Fifth IEEE International Conference on. IEEE, 2007.

[8] Crocker, David, and Judith Carlton. "Verification of C programs using automated reasoning." Software Engineering and Formal Methods, 2007. SEFM 2007. Fifth IEEE International Conference on. IEEE, 2007.

[9] RTCA. DO-332 Object-Oriented Technology and Related Techniques Supplement to DO-178C and DO-278A, RTCA, 2011.

[10] Liskov, Barbara H., and Jeannette M. Wing. "A behavioral notion of subtyping." ACM Transactions on Programming Languages and Systems (TOPLAS) 16.6 (1994): 1811-1841.

[11] Dos Reis & Stroustrup. “Specifying C++ concepts”, Dos Reis, Gabriel, and Bjarne Stroustrup. ACM SIGPLAN Notices 41.1 (2006): 295-308.

[12] Batty, Mark, et al. "Mathematizing C++ concurrency." ACM SIGPLAN Notices. Vol. 46. No. 1. ACM, 2011.

[13] IEEE 754-2008, “IEEE Standard for Floating-Point Arithmetic”, ISBN 978-0-7381-5752-8, 2008.

[14] Leavens, Gary T. "An overview of Larch/C++: Behavioral specifications for C++ modules." Object-Oriented Behavioral Specifications. Springer US, 1996. 121-142.

[15] Moy, Yannick, and Claude Marché. "The Jessie plugin for Deduction Verification in Frama-C—Tutorial and Reference Manual. INRIA & LRI, 2011."

[16] Dahlweid, Markus, et al. "VCC: Contract-based modular verification of concurrent C." Software Engineering-Companion Volume, 2009. ICSE-Companion 2009. 31st International Conference on. IEEE, 2009.

[17] Carlton, Judith, and David Crocker. "Escher Verification Studio: Perfect Developer and Escher C Verifier." Industrial Use of Formal Methods: Formal Verification: 155-193, 2013. ISBN 13: 9781848213630

[18] Jacobs, Bart, et al. "VeriFast: A powerful, sound, predictable, fast verifier for C and Java." NASA Formal Methods. Springer Berlin Heidelberg, 2011. 41-55.

[19] Rainer-Harbach, Marian. "Methods and Tools for the Formal Verification of Software", Technische Universität Wien, 2011. Retrieved from http://aragorn.ads.tuwien.ac.at/publications/bib/pdf/rainer-harbach_11.pdf, 12 June 2014.

[20] Crocker, David. "Safe object-oriented software: the verified design-by-contract paradigm." Proceedings of the Twelfth Safety-Critical Systems Symposium (ed. F.Redmill & T.Anderson) 19-41, Springer-Verlag, London, 2004. ISBN 1-85233-800-8

[21] Spark 2014 Reference Manual, section 7.1.2. Retrieved from http://docs.adacore.com/spark2014-docs/html/lrm/packages.html#external-state, 31 August 2014.

[22] http://www.ncl.ac.uk/computing/research/project/4519, retrieved 31 August 2014.