Verification of C Programs Using Automated Reasoning

David Crocker and Judith Carlton
Escher Technologies Ltd.
Mallard House, Hillside Road, Ash Vale,
Aldershot GU12 5BJ, United Kingdom
{dcrocker, jcarlton}@eschertech.com

Abstract

Much of the embedded software development market has necessarily tight constraints on program size and processor power, hence developers use handwritten C rather than autocode. They rely primarily on testing to find errors in their code.

We have an established software development tool known commercially as Perfect Developer, which uses a powerful automatic theorem prover and inference engine to reason about requirements and specifications. We have found that automated reasoning can be used to discharge a very high proportion of verification conditions arising from the specification and refinement of software components described in our formal specification language, Perfect. The Perfect Developer tool set can also generate code in a C++ subset or in Java, and the output code is then virtually certain to meet the stated specification, reducing the need for exhaustive testing. However, this is not helpful to developers of embedded software who are constrained to write code by hand.

We therefore decided to investigate whether automated reasoning could provide a similar degree of success in the verification of annotated C code. We present our preliminary findings.

1. Introduction

Software is being increasingly used in safety- and mission-critical applications. Many of these applications demand a degree of dependability that is difficult to achieve using traditional software development techniques. It is often impossible to demonstrate that the required integrity level has been achieved through testing alone, due to the limited number of tests that can be performed, even if the testing phase spans many weeks. Therefore, it is preferable to develop critical applications using a correctness-by-construction approach that eliminates most sources of error well before testing commences. A vital part of this approach is formal proof that execution of the software will meet the stated requirements, together with the implicit requirements that the software will not crash or suffer other runtime errors such as divide-by zero or buffer overflow. Testing is still used, but its primary role is to check that the construction method and tool chain are sound.

Correctness-by-construction starts with a formal specification of the requirements. Some practitioners use a specification language such as Z or VDM for this purpose, in conjunction with a traditional programming language for implementation. Our preferred approach is to use a single formal notation that provides for specification, design and refinement, followed by automatic translation of the most refined version of the design to ready-to-compile code. Examples of systems that support this approach are B [2] and our own *Perfect Developer* [11].

In some sectors of the embedded software market, for example, in high-volume consumer electronics, processor cost is critical, so available processor power is tightly constrained. In space applications, electrical power is at a premium, so once again processor power is limited. In applications such as these, handwritten code is preferred to generated code because of its compactness and efficiency. Even so, system size is increasing, which means that verification by exhaustive testing is becoming less and less practicable.

Our success in using automated reasoning to discharge a very high proportion of generated verification conditions (also called *proof obligations*) in *Perfect Developer* led us to consider whether we could adapt the tool to achieve the same success rate in attempting to verify programs written in C, a popular implementation language for embedded software. The approach we took was to augment the tool with a front-end for processing a subset of C. We were able to represent all the types supported by C as *Perfect* types with only minor extensions. Similarly, we found we could translate most statement types supported by the C language into statements supported by the refinement sublanguage of *Perfect*. The major extension we added to the semantic proces-

sor was support for expressions with side-effects.

2. Related Work

The approach of annotating programming languages with specification constructs has been described by a number of authors. Current implementations of annotated development fall into two camps.

The first camp comprises tools aimed at providing proof of correctness. These tools typically enforce a strict subset of the selected programming language and are therefore better suited to new development of safety- and mission-critical software rather than trying to improve the quality of legacy code. The most successful example of this genre to date is SPARK Ada [12], a commercially-supported notation that has been used to develop some sizeable critical systems. More recent tools include Spec# [5] which adds contracts and other specifications to C#, and Why [15], which has been used to verify programs in a number of languages including C [16] using both automatic and human-assisted proof tools.

The second camp comprises tools whose goals are ease of use and the detection of a substantial proportion of bugs - such as buffer overflow and other run-time errors - rather than proving program correctness. Examples of these 'lightweight' tools include Splint [13] (a derivative of Larch [9]) and ESC/Java [6]. Some of these tools intentionally sacrifice completeness and soundness in order to be more useful on un-annotated or lightly-annotated code.

Also useful in this space is is Abstract Interpretation [1], a technique for checking that programs are free from run-time errors that does not require the source code to be annotated. Although sound, it uses conservative approximations of the state space to keep the problem size manageable, at the expense of sometimes reporting potential errors that in reality cannot occur.

Our own work is focussed on proof of correctness - not just freedom from runtime errors - for critical software systems using a fully automatic prover. Systems that employ interactive proof assistants have the advantage that even difficult proofs can be discharged with enough manual effort; but our view is that only systems that use fully automatic provers stand any chance of widespread industrial acceptance.

3. Towards a verifiable C

In the C language, it is easy to write constructs that have undefined or implementation-defined behaviour. It follows that C is far from an ideal choice for writing critical applications. However, the C language definition documents [3, 4] state fairly clearly the situations in which behaviour is not well-defined. This makes it possible to define subsets of C with well-defined semantics. One attempt at such a subset is MISRA C [10].

Although many of the restrictions imposed by such subsets can be enforced statically, others depend on the interrelation between different parts of the program, in ways that can only be resolved using mathematical proof.

Whereas some of the MISRA rules are based on notions of "good programming practice" rather than the avoidance of undefined behaviour, we have selected our subset of C solely with verification in mind. Not surprisingly, almost all the constructs that are prohibited in our subset are also prohibited in the MISRA subset, although the reverse is not true.

In adapting *Perfect Developer* for C, we chose to retain the Verified Design-by-Contract paradigm [14] that the tool uses when verifying *Perfect*. This paradigm requires as a minimum that all function preconditions must be explicitly stated. Although this may seem rather burdensome for the programmer, the rigorous documentation of preconditions greatly enhances maintainability and reusability, even when formal methods are not otherwise used. In a critical system, we maintain that when a programmer writes a function call, there should be no uncertainty in his/her mind as to what conditions must be satisfied in order that the called function can be relied upon to behave as expected.

The usual way of augmenting programs with formal specifications is to express them as specially-formatted comments. In many languages, no other choice is available, assuming the requirement that the source text is to be compiled by a standard, unmodified compiler. However, formatting specifications as comments regrettably suggests to developers that they are unimportant to the functioning of the program. We prefer specifications to have a more central role in the language. Fortunately, the C language offers us an alternative: namely, to define specification constructs using macros. In C, macros can be defined so as to expand to nothing when the program is compiled, so that the specification is invisible to a standard compiler but visible to our verifier. For example, we declare function preconditions using the construct **pre**(*expression*), having made the macro definition:

```
#define pre(x) /* nothing */
```

visible to the compiler.

One small disadvantage of using macros is that under C89 the number of arguments must be fixed for any particular macro. If the user tries to declare two comma-separated preconditions in a single **pre** clause like this:

```
pre(p1, p2)
```

then the compiler will refuse to compile the program because it is expecting a single macro argument but finds two. However, the following alternative forms:

```
pre(p1) pre(p2)
pre(p1; p2)
pre((p1, p2))
pre(p1 && p2)
```

all avoid this problem.

4. Applying Design-by-Contract to C

Any attempt to define contracts for functions written in C must address the particular challenges that this language presents. Some of these are discussed here. We use the term *ghost* to refer to an entity that can be used in a specification construct but not in the program text visible to the compiler.

4.1. Arrays and pointers

Possibly the biggest flaw in the C language is its lack of distinction between a pointer to a single storage location and an array (which, in C, is just a pointer to a block of contiguous locations). A related problem is that when an array is passed as a parameter to a function, the called function has no way of determining the number of addressable elements in the array, unless this information is passed in an additional parameter. However, when writing specifications, it is frequently necessary to refer to the length of an array.

We work round this problem as follows. First, we distinguish between pointers to single locations and pointers to arrays. This we do by using the keyword *array* when declaring an array pointer. Thus, in the following declarations:

```
int *p;
int array *pa;
```

the variable *p* is a pointer to a single element, whereas the variable *pa* is a pointer to an array. Our processor enforces the correct use of this annotation, for examply by prohibiting the use of pointer arithmetic and indexing on plain pointers.

Second, we treat an array pointer as if it is a structure comprising three variables: two ghost integer variables named *lwb* and *upb* which represent the lower and upper bounds of the array, and the storage pointer itself. The main use of the ghost variables is to describe the function preconditions that ensure array accesses are in bounds. For example, the precondition of the array access `pa[i]` is $pa.lwb <= i \ \&\& \ i <= pa.upb$. As a further convenience, we provide an additional ghost member *lim* (short for limit) defined as *upb+1*.

A further consideration is that whereas C allows the value of any pointer to be zero (i.e. pointing to nothing), in many contexts a zero value is inappropriate, because any attempt to dereference it will lead to undefined behaviour. We require the programmer to state explicitly that particular variables and parameters are allowed to take null values. We do this by decorating their declarations with the keyword *null*. In the absence of such decoration, zero-valued pointers are prohibited. For example, in the following declarations:

```
int null array *pna;
int *p;
```

variable *pna* may point to an array or be zero, while variable *p* is a nonzero pointer to a simple variable or single element. This approach avoids the need to pepper the program with preconditions of the form `p != 0`. Furthermore, when a pointer is dereferenced, we do not need to generate a verification condition that it is non-null unless the user has declared that null is a permitted value for it.

Next, the correct operation of many functions (including library functions like *strcpy*) requires that two or more array pointers passed into the function refer to non-overlapping storage. We allow this to be expressed by augmenting our representation of pointers and array pointers with a ghost member function *disjoint*, which takes another pointer as a parameter and returns true if and only if the result of any legal indexing of the pointers cannot refer to the same element. Of course, C has no notion of functions that are structure members, but we felt here that it was sensible to borrow a small amount of syntax from C++.

Finally, a string literal in C has three different meanings depending on whether it is used as an array initialiser, the operand of `sizeof`, or in a context where a value of type `char*` (actually `const array char*` in our subset) is required. We handle this by assigning the *Perfect* type *seq of char* to string literals, but allowing automatic type conversion of string literals to `const array char*` where required. A further refinement allows for the fact that compilers are permitted to perform string pooling, so that identical string literals appearing in different places in the source code are not necessarily disjoint when converted to array pointers.

4.2. Unions

Although coding guidelines for critical systems often deprecate the use of undiscriminated unions such as are

supported in C, we feel that in the absence of alternative facilities (such as discriminated unions, or inheritance and polymorphism), unions are too useful and should be tamed rather than ignored. In our semantic model, each variable of union type remembers the name of the union member through which its current value was assigned. When a member of a union variable is read, it is a precondition that the member being read must match the remembered name. The remembered name can be queried using the ghost expression `v holds member` where *v* is an expression of union type and *member* is the name of one of the members of its type.

4.3. Side effects

In writing specifications, it is important to use expressions that are free from side-effects. Unfortunately, side-effects are common in C expressions. We use the term *pure* to indicate that an expression is side-effect free, and a function can be annotated with this keyword to indicate that it has no side-effects. The language processor enforces correct use of this keyword in a conservative manner by reporting an error if the function is found to modify any global variables, or if it writes through any pointers, or if it calls another function that is not annotated *pure*. All expressions used in specifications are required to be *pure*.

4.4. Quantifiers

The expression sublanguage of C does not provide quantification over types or collections. However, quantification is frequently needed in specifications. We therefore imported quantified expressions from *Perfect* with syntactic modifications to fit better with the style of C. We allow quantification over types and over ranges of integers, using the expression `a..b` to denote the set of integers from *a* to *b* inclusive (or the empty set if $b < a$). For example, the expression:

```
forall int i in a.lwb..a.upb:- a[i]>0
```

expresses the notion that all members of the array *a* are positive. These expressions can, of course, be used only within specifications.

5. A small example

Consider a function that uses the *strcat* and *strcpy* functions from the Standard C Library to build a message of the form "Error *type* at *location*" in a buffer supplied by the caller, where *type* is an indexed member of list of error messages and *location* is an input string. This program could exhibit various run-time errors including the following:

- If *type* is not a valid index into the list of messages, the behaviour is undefined
- If either the input string or the error message is not null-terminated, characters will be read beyond its upper bound
- If the buffer is too short to contain both strings, it will overflow
- If either string is a null pointer, the program may crash
- If the output buffer overlaps the input string, the copy operation may overwrite the input data before it has all been read, overwriting the terminating null and resulting in a nonterminating copy operation and buffer overflow

We wish to specify and write this function in such a way that correct operation is assured as long as a declared precondition is satisfied by the caller. In turn, we will verify that all callers of the function satisfy the precondition.

First we declare an auxiliary ghost function *isNullTerminated* so that we can more easily describe null-terminated strings. Next, we augment the *strcpy*, *strcat* and *strlen* function prototypes with specifications. In order to avoid the need to modify the standard C header file *string.h* we provide a mechanism for declaring a function specification separately from its normal prototype declaration, using the keyword **spec** to indicate that a declaration is provided for specification purposes only and it should be attached to a function or function prototype declared elsewhere. For convenience, we package these and many other specifications of C library functions in a special header file *arc.h*. Extracts from this file relevant to our example are shown in Listing 1.

We can now write the function itself, together with some example client code (Listing 2). This example give rise to 26 verification conditions, a surprisingly high number for such a small example. Of these, 6 relate to an index into *errorMessages* being in bounds, another 6 relate to the preconditions of *buildMessage* being satisfied at its point of call, and no less than 14 relate to the preconditions of *strlen*, *strcpy* and *strcat*.

6. Beyond freedom from run-time errors

The preceding example demonstrates a level of function specification sufficient to ensure absence of undefined behaviour and run-time errors. However, with a little more effort we can add postconditions that specify the required outcomes, and then use the verifier to prove that they will be achieved. Listing 3 shows a binary search function, augmented not only with the preconditions needed to ensure correct behaviour, but also with postconditions to define the required result, a loop variant (the **decrease** clause)

```
#if defined(__ARC__) && !defined(__ARC_H_INCLUDED__)
#define __ARC_H_INCLUDED__

// Specify the standard string functions for the verifier

ghost pure bool isNullTerminated(const array char* s)
    post (result == (exists i in 0..s.upb :- s[i] == '\0'));

spec int strlen(const array char* src)
    pre (isNullTerminated(src))
    post (result in 0..src.upb && src[result] == '\0'
        && (forall j in 0..(result - 1) :- src[j] != '\0'));

spec array char* strcpy(array char* dst, const array char* src)
    pre (isNullTerminated(src); dst.disjoint(src); dst.upb > strlen(src))
    post (src.elems == old src.elems; isNullTerminated(dst); strlen(dst) == strlen(old src));

spec array char* strcat(array char* dst, const array char* src)
    pre (isNullTerminated(src); isNullTerminated(dst); dst.disjoint(src);
        dst.lim > strlen(src) + strlen(dst))
    post (src.elems == old src.elems; isNullTerminated(dst);
        strlen(dst) == strlen(old dst) + strlen(src));

#else

// Define the specification macros as expanding to nothing when the program is compiled

#define array
#define assert
#define change(x)
#define decrease(x)
#define ghost
#define keep(x)
#define null
#define post(x)
#define pre(x)
#define pure

#endif
```

Listing 1. Extract from file arc.h

```
#include "arc.h"
#include <string.h>

const int systemError = 0, inputError = 1, commandError = 2;
static const char array * errorMessages[] = {"System", "Input", "Command"};

size_t buildMessage(int type, const array char* location, array char* dst)
pre (0 <= type; type <= errorMessages.upb)
pre (isNullTerminated(location))
pre (dst.disjoint(location); dst.disjoint(errorMessages[type]))
pre (strlen(location) + 10 + strlen(errorMessages[type]) <= dst.upb)
{
    strcpy(dst, errorMessages[type]);
    strcat(dst, " error in ");
    strcat(dst, location);
    return strlen(dst);
}

size_t test()
{
    char buffer[50];
    return buildMessage(inputError, "I/O module", buffer);
}
```

Listing 2. Error message function and client code

to facilitate proof of termination, and loop invariants (the **keep** clause) to justify the correctness of the loop. The 34 verification conditions arising from this example are discharged by the automated prover in less than 30 seconds on a personal computer of modest performance. Of these verification conditions, 15 relate to operator preconditions (mostly index-in-bounds), 6 to correct initialisation of the loop, and 9 to correct behaviour of the loop. The remaining 4 demonstrate that the return value of the function satisfies its specification. The proofs can be viewed at `http://www.eschertech.com/arcpaper/proofs`.

7. Help with annotations

In our previous experience with *Perfect Developer*, we have observed that users quite often miss out trivial preconditions and loop invariants, such as conditions for array indices to be in bounds. This led us to enhance the verifier so as to produce suggested amendments to the specification, when proofs cannot be found and certain conditions are met. For example, if an unproven verification condition involves only the values of inputs to a function, it is very likely that the user forgot to state the required condition as a precondition of the function; so the verifier will suggest it as an additional precondition.

This mechanism is also operational when we use the verifier on C programs. If, in our first example, we remove the preconditions from the declaration of function *buildMessage*, the verifier makes a number of suggestions including the following:

Add extra precondition: 0 <= type
Add extra precondition: type <= 2
Add extra precondition: isNullTerminated(location)

It is harder for the verifier to make useful suggestions in more complex cases, for instance if we remove some of the loop invariants from our second example.

8. Conclusions and further work

In this paper we have demonstrated the annotation of two small C programs with specifications, in order that they may be verified with respect to freedom from run-time errors and (in the second example) with respect to a stated requirement. The binary search example is not trivial, but such is the state-of-the-art in automated reasoning technology that our prover had no trouble in generating all the necessary proofs without user intervention (our prover is non-interactive by design).

We have found that applying the Design-by-Contract paradigm to C programs requires a substantial amount of annotation. Whereas SPARK starts from a well-designed language (i.e. Ada) which it subsets in the interest of verifi-

```
#include "arc.h"

// Find the index of the lowest element that compares with or above the parameter
int search(const array double *table, int size, double x)
pre (size == table.lim)
pre (forall i in 0..table.upb; j in 0..table.upb :- !(i >= j) || table[i] >= table[j])
post (result >= 0;
        result <= size;
        forall z in 0..(result - 1) :- x > table[z];
        forall z in result..table.upb :- x <= table[z])
{
    int i = 0, k = size;
    while (i != k)
        change (i; k)
        keep (0 <= i; i <= k; k <= table.lim)
        keep (forall z in 0..(i - 1) :- x > table[z];
                forall z in k..table.upb :- x <= table[z])
        decrease (k - i)
    {
        int mid = (i + k)/2;
        if (x > table[mid])
        {
            i = mid + 1;
        }
        else
        {
            k = mid;
        }
    }
    return i;
}
```

Listing 3. Implementation of a binary search algorithm

ability, and the notation of our own *Perfect Developer* was specifically designed for safety and verifiability, the C language is in comparison poorly suited to verification. However, by the careful introduction of additional keywords such as **array** and **null** to mitigate weaknesses of C, we have reduced the volume of preconditions and other specification annotations required in order to make verification possible while still allowing the use of a standard compiler. Thus, although we consider it preferable to use SPARK or *Perfect Developer* in language-agnostic situations, automating the generation of correctness proofs does appear to be possible where practical considerations dictate the use of C. In some cases, missing annotations can be suggested automatically by the verifier; however manual review of such suggestions is advisable.

The Verified Design-by-Contract paradigm works well for functions that are exposed to external callers, as they permit verification of the called function to be done independently of verification of the caller. However, the benefits of annotating simple functions that cannot be called externally, such as C functions with `static` scope, are less clear. One possibility is to "inline" such functions at the points at which they are called, so that they do not require specification.

A significant problem with verification of software written in conventional programming languages is the need for loop invariants. Some invariants for simple loops can be deduced fairly easily - for example, those needed to ensure that array indices are within bounds - but determining all the loop invariants required for more complex cases such as our binary search example remains a challenge. Our preferred approach is to provide higher-level constructs to avoid the need to hand-code loops. For example, we find that in software written in *Perfect*, only 8% of loops in the final code are derived from explicit loops in the source, with the remainder being generated from constructs such as quantified postconditions. This is of no use when the programmer is constrained to write in a lower-level language. However, some work has been done on the automatic generation of loop invariants [7], [8] and we hope to apply this to our own research in the future.

References

[1] P. Cousot and R. Cousot, *Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints* (ACM POPL 1977).

[2] J-R. Abrial, *The B-Book: Assigning Programs to Meanings* (Cambridge University Press, 1996).

[3] *American National Standard for Information Systems - Programming Language C, ANSI X3.159-1989* (American National Standards Institute, 1989).

[4] *ISO/IEC 9899-1999, Programming Languages - C* (International Standards Organization, 1999).

[5] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte, *The Spec# programming system: An overview* (CASSIS 2004, LNCS vol. 3362, Springer, 2004).

[6] K. Rustan M. Leino, Greg Nelson, and James B. Saxe, *ESC/Java User's Manual* (Technical Note 2000-002, Compaq Systems Research Center, October 2000).

[7] Jamie Stark and Andrew Ireland, *Invariant Discovery via Failed Proof Attempts* (Lecture Notes in Computer Science 1559, 271-288, 1999).

[8] K. Rustan M. Leino, Francesco Logozzo, *Loop Invariants on Demand* (APLAS 2005, 119-134).

[9] J. V. Guttag and J. J. Horning, *Introduction to LCL, A Larch/C Interface Language* (available at http://ftp.digital.com/pub/Compaq/SRC/research-reports/abstracts/src-rr-074.html).

[10] *MISRA-C:2004 - Guidelines for the use of the C language in critical systems* (Motor Industry Software Reliability Association 2004, ISBN 0 9524156 2 3).

[11] D. Crocker and J. Carlton, *A High Productivity Tool for Formally Verifed Software Development* (Escher Technologies, 2004. Available at http://www.eschertech.com/papers/pdpaper.pdf).

[12] J. Barnes, *High Integrity Software: The SPARK Approach to Safety and Security* (Addison Wesley 2003, ISBN 0-321-13616-0).

[13] David Evans and David Larochelle, *Improving Security Using Extensible Lightweight Static Analysis* (IEEE Software, Jan/Feb 2002).

[14] David Crocker, *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).

[15] J.-C. Fillitre, *Why: a multi-language multi-prover verification tool* (Research Report 1366, LRI, Universit Paris Sud, 2003).

[16] Jean-Christophe Fillitre and Claude March, *Multi-Prover Verification of C Programs* (Lecture Notes in Computer Science 3308, 15-29, 2004).