Deductive Functional Verification of Safety-Critical Embedded C-Code: An Experience Report

Dilian Gurov[1], Christian Lidström[2(✉)], Mattias Nyberg[1,2], and Jonas Westman[1,2]

[1] KTH Royal Institute of Technology, Stockholm, Sweden
[2] Systems Development Division, Scania AB, Södertälje, Sweden
christian.lidstrom@scania.com

Abstract. This paper summarizes our experiences from an exercise in deductive verification of functional properties of automotive embedded C-code in an industrial setting. We propose a formal requirements model that supports the way C-code requirements are currently written at Scania. We describe our work, for a safety-critical module of an embedded system, on formalizing its functional requirements and verifying its C-code implementation by means of VCC, an established tool for deductive verification. We describe the obstacles we encountered, and discuss the automation of the specification and annotation effort as a prerequisite for integrating this technology into the embedded software design process.

1 Introduction

While Formal Methods are in general only slowly making their way into industrial practice for quality assurance, their adoption in the domain of embedded, safety-critical systems has seen much progress over the last years. One reason for this development, from an industry perspective, is the increased analyses effort advocated by standards to achieve functional safety of such systems. For example, automotive functional safety standard ISO 26262 recommends formal verification for higher levels of criticality. The smaller size of embedded code as compared to arbitrary applications, and the constraints on how code is structured in order to safeguard against potential unwanted behaviours, are also enabling factors for the application of the typically more expensive formal analysis techniques.

Scania is a leading manufacturer of commercial vehicles, and specifically heavy trucks and buses. A large part of the embedded C-code developed at Scania is safety-critical, and a considerable effort is spent during code development and deployment on quality assurance. On top of the traditional testing methods, Scania is exploring the possibility for integrating deductive verification and model checking into the code design and quality assurance process.

Work partially funded by Vinnova within the KLOSS AkUt initiative, which sent academics out to Industry one day a week for half a year during 2015/2016.

L. Petrucci et al. (Eds.): FMICS-AVoCS 2017, LNCS 10471, pp. 3–18, 2017.
DOI: 10.1007/978-3-319-67113-0_1

The main motivation for this are the increased safety requirements resulting from innovative trucking solutions such as platooning and autonomous driving.

Motivating Factors. The starting point for the work described here were the following general findings and concrete observations made from studying a particular C-module and its associated requirements document [1].

1. Many of the requirements of the module are *functional*, in the sense that they express output values as a function of input values (i.e., as a mathematical function). A common case is that two or more outputs of the same module depend on intersecting sets of inputs. This leads to a natural *functional decomposition* that allows the functionality of modules to be understood conceptually through the metaphor of a *combinational logic circuit*.
2. The well-known and established logic and deductive system called *Hoare logic* has been developed precisely for formally specifying and proving this type of properties [12]. Program verification in this style is based on *logical assertions*, which are essentially state properties expressed as first-order formulas over program and (additional) logical variables, typically in the form of pre- and postconditions to C-functions, or loop invariants. The assertions are tied to specific control points of the program, usually by means of program annotations provided by the programmer. The annotated program is then translated by purely symbolic means (based on computation of so-called weakest preconditions, or on symbolic execution) to a first-order logic formula, called verification condition, which is true exactly when the assertions hold for the annotated program. The generated verification condition is then passed on to an automated (back-end) theorem prover to be checked for validity.
3. The typical control flow of embedded code, and the used datatypes, follow certain constraints, described through guidelines following MISRA C and ISO 26262, which render the given correctness problem *decidable*. For instance, most of the code we examined does not involve any looping constructs, which typically require loop invariants to be provided by the programmer, and most of the data is of enumerable (i.e., non-inductive) types, and thus no datatype invariants need to be provided.
4. There exist mature tools such as VCC [4,5] and Frama-C [8] that support the automated deductive verification of C-code supplied with annotations.

All these observations and findings were a strong indication that the formal specification of requirements and their deductive functional verification can be automated to a degree that makes them a viable option for increased quality assurance of the safety-critical embedded C-code. This lead to the present pre-study, which builds on the findings of two Master theses [10,13].

Goals of the Study. The main question that the present pre-study addresses is: is it feasible, and what would be needed, to push formal requirement specification and deductive functional verification into an embedded software design process, such as the one at Scania?

Our concrete questions concern:

1. Formalization of functional requirements: how user-friendly can it be made, and how much effort does it take?
2. Verification tool: what is the code coverage regarding the given code base, how easy is it to use the tool and to make sense and use of the feedback it provides on failed verifications, and how efficient is it in practice?
3. Annotation of the code: how much effort does it take, what annotation overhead does this incur, and how automatable is the annotation process?

Structure of the Paper. The remainder of the paper is organized as follows. In Sect. 2 we describe the type of requirements we found in the requirements document, and propose a formal requirements model, based on mathematical functions, for capturing such requirements. In Sect. 3 we describe the verification tool VCC, the verification method it supports, and the specifics of its use. In Sect. 4 we discuss (without, however, revealing proprietary code or information) the module which we considered in our pre-study, its requirements, the annotation process, and the obstacles which we encountered. Our findings are summarized in Sect. 5, together with a proposal for a semi-automated specification and verification process based on these findings. Related work we describe in Sect. 6, and conclude with Sect. 7.

2 Formalizing Functional Requirements

In this section we describe the character of the requirements as we encountered in the requirements document of the module we considered [1], and propose a formal requirements model.

Requirements. The requirements in the provided requirements document are written in terms of a set of *requirements variables*, which are (model) variables distinct from the program variables. This follows a clean discipline of separating specifications from implementations.

A significant number of the requirements are presented in a format illustrated by the following concrete example:

```
While SecondaryCircuitHandlesSteering == True
   If ParkingBrakeSwitch == ParkingBrakeNotSet
      ElectricMotor = On
```

which could be described as a *conditional assignment form*. On its own, such a requirement does not specify completely the value of the variable being set (here, `ElectricMotor`). Since the same variable may be assigned a value by more than one requirement, this immediately raises the questions of whether the set of requirements is *complete* (i.e., it specifies, for all values of the input variables, a value for every output variable) and whether it is *consistent* (i.e., specifies at most one such value), together guaranteeing the well-definedness of

the specified data transformation. While specifications may be incomplete by intention, inconsistency is always a problem that needs to be resolved.

Further noteworthy to observe is that, while some of the requirements variables used in the specifications do correspond to global *module interface* ones (i.e., variables through which the module interacts with its environment), most do not; instead they are *intermediate* requirements variables. This corresponds conceptually to a *function decomposition* of the functions computed by the module. Such a break-down of requirements constitutes a natural representation of a multi-output function when its outputs depend on intersecting sets of inputs, and makes the reading of requirements easier. It also allows the functionality of modules to be understood conceptually, and visualized, through the metaphor of a (multiple-valued) *combinational logic circuit*.

Many of the intermediate requirements variables have corresponding counterparts in the code in the form of local variables or struct fields (sometimes even more than one, as certain values are transferred by reference via calls to helper functions). These observations raise the question whether one should aim to verify every requirement individually, or only the induced functional dependence of the output variables on the input ones. The first option can only be realized by referring in the code annotations to the local code artifacts, and thus ties the verification to the implementation. This goes against the principle of separating specifications from implementation details, which allows the implementation to evolve without necessarily changing the requirements. On the other hand, the second option may result in considerably worse verification times, as it is usually the case when verifying a specification in a "black-box" manner, not utilizing the implementation information.

Formal Requirements Model. To formalize the requirements, one has first to define a *formal requirements model*. In the present case of purely functional data transformation, it is suitable to base the formal model on the standard discrete-mathematical notion of *(partial) function*.

As an example, consider variables $x_1, x_2 \in \{7, -3\}$ and $x_3, y \in \{-2, 6\}$, and let the value of variable y depend functionally on the values of the variables x_1, x_2 and x_3, as defined by the table on the left of Fig. 1. We shall use this (rather trivial) example to discuss possible presentations of such functions.

Function Views. One can distinguish between two views on mathematical functions. First, there is the *black-box* view, which describes the functions computed by a module via *interface variables* only, i.e., as module *contracts*. This view is important for *modular verification* (say, in an assume/guarantee style), as it is the view of the module that is exported to the rest of the system. In principle, this is the view to be verified, since it specifies just the data transformation to be computed by a module and nothing more. And then, there is the *white-box* view, which decomposes the functions by introducing *intermediate variables*. This view is important for readability and simplicity of module specifications. However, as explained above, the verification of the individual requirements resulting from the breakdown is problematic. Ideally, intermediate variables should only be

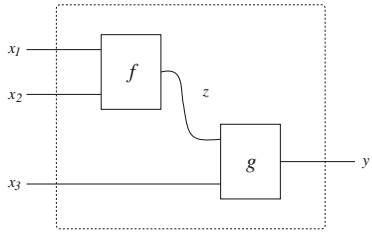
x_1	x_2	x_3	y
7	7	−2	6
−3	7	−2	6
7	−3	−2	6
−3	−3	−2	−2
7	7	6	−2
−3	7	6	−2
7	−3	6	−2
−3	−3	6	6

(a) Function.

x_1	x_2	f
7	7	7
−3	7	7
7	−3	7
−3	−3	−3

z	x_3	g
7	−2	6
−3	−2	−2
7	6	−2
−3	6	6

(b) Function decomposition.



(c) Function architecture.

Fig. 1. A function and its decomposition.

used as a vehicle to relate output values to input values. The two views have a simple mathematical connection by means of *function decomposition* in the one direction, and *function substitution* in the other.

For instance, the example function defined above can be decomposed according to the equations:

$$z = f(x_1, x_2)$$
$$y = g(z, x_3)$$

introducing the intermediate requirements variable $z \in \{7, -3\}$ and the functions f and g defined by the tables in the middle of Fig. 1. The "architecture" of this decomposition, or white-box view of the function, is depicted on the right of Fig. 1, in the style of a combinational logic circuit.

3 The Verification Tool VCC

VCC, standing for Verified Concurrent C, is a tool for the formal verification of programs written in the C language [5]. As the name suggests, it supports verification of concurrent code. It has been developed at Microsoft Research, and is available for Windows under the MIT license via GitHub[1].

The assertions to be verified, such as *function contracts* and *data invariants*, are to be specified by the programmer directly in the C source code in the form of *annotations*. VCC has its own syntax for this: annotations are always enclosed in parentheses and preceded by an underscore '`_(...)`', but otherwise follow a syntax similar to the one of the C language itself.

The contract of a function is a set of annotations located between the function header and its body. The set typically includes a *precondition* expressing an assumption on the values of the actual parameters and global variables at the time of invoking the function, and a *postcondition* relating the return value and the values of the global variables at the time of returning from the function

[1] See `github.com/Microsoft/vcc`.

```
#include <vcc.h>

void swap(int *p, int *q)
  _(writes p,q)
  _(ensures *p == \old(*q) && *q == \old(*p))
{
  int tmp;
  tmp = *p;
  *p = *q;
  *q = tmp;
}
```

Fig. 2. A simple C-function annotated with a contract.

call to the former values. The remaining annotations essentially specify other side-effects of executing the function body.

An example of a contract for a simple C function is given in Fig. 2. The header file `vcc.h` is included in order to make the compiler ignore the VCC annotations. In the contract, the `ensures` clause specifies a postcondition, where an expression appearing within an `\old` clause refers to its value at function invocation time (for preconditions this is the default mode). The postcondition states that the value pointed to by `p` upon return from the method call will equal the value pointed to by `q` before execution, and *vice versa*. Preconditions are specified with the keyword `requires`. There is no precondition provided in this specification, meaning that the contract should hold for any values of the actual parameters and global variables at the time of invoking the function. The `writes` clause specifies the side-effect that both argument pointers are writable, and no other memory locations. It also serves to give notice to any calling function that (only) the contents of the specified memory locations may change during the call. In contrast, the lack of a `writes` clause tells the caller that this function will not have any visible side-effects w.r.t. the specified variables.

Contracts can be specified not only for functions, but for any block of code. Assertions can be inserted at any control point of executable code, and are useful both to provide hints to VCC and for troubleshooting. Finally, one can specify invariants for both loops and data structures.

Verification Method. VCC is a deductive verifier. The annotations are translated into an intermediate language and provided to another tool, Boogie, from which proof obligations are generated; these are then discharged by an automated theorem prover, the default being Z3.

The verification of function contracts is *modular*: when checking the body of a function, and a call to another function is encountered, the tool asserts that the caller fulfills the preconditions of the callee, and assumes that the callee's postconditions hold right after the call statement. This function modularity ensures scalability of the verification method w.r.t. the number of C functions.

The verification performed by VCC is claimed to be *sound*, in the sense that verified assertions do indeed hold, but is not guaranteed to be complete, meaning that assertions that could not be verified may still hold. Sometimes the programmer can "help" the tool by rewriting assertions to equivalent formulas that can be handled by the back-end reasoning engine.

Ghost Code. During verification, VCC keeps track of an internal state referred to as the *ghost* state [5]. Apart from logical representations of all actual program variables, this state also includes many other abstract data structures and functions that are needed to provide a model in which to reason about the program. In addition to function contracts, VCC provides numerous other ways of manipulating the ghost state, allowing the programmer to assist the reasoning engine in performing a successful verification.

Ghost functions can be defined with the keyword `def`. Such functions must have no side effects, and may only be used in specifications. Also regular (but side-effect free) C functions can be marked with the keyword `pure` to allow them to be used in specifications. Ghost variables are declared with the keyword `ghost` preceding an ordinary C declaration. For ghost variables, any native or user created C type can be used, as well as a number of types built into VCC. For example, there are mathematical integers (`\integer`), natural numbers (`\natural`), and true Booleans (`\bool`), to name a few.

Memory Model. C is often referred to as a low-level programming language, because of the similarity between its primitive objects and those of hardware. In addition, C has explicit memory (de)allocation, pointer arithmetic and aliasing, and a weak type system that can be easily circumvented, all of which makes reasoning about memory harder. VCC, however, has a stricter memory model and stronger typing for its ghost state [7]. System memory is represented as a set of typed objects, and is maintained in the ghost memory as pointers to all valid objects. One guarantee of this model is that valid objects are always separated. VCC can thus efficiently take advantage of well-written C code and elevate it to its own stronger model. In cases where it is not able to do this, verification will fail, and additional annotations regarding the usage of memory are needed.

Because of its focus on concurrent code, ownership and closedness information is also stored for each object in ghost memory [6]. For example, threads are only allowed to make sequential writes to memory of which they are the owner, and sequential reads to memory that they own or can be proved not to change. Ownership is represented as a tree structure. The system heap is organized as a set of trees, where each root node represents a thread and each edge represents ownership by the source node. A thread is the direct owner of its local variables, whereas a struct owns its fields and is itself owned by some higher-level object.

4 The Case Study

Our case study is based on a C-code module that is part of the embedded system controlling the Scania trucks, and is considered safety-critical. More specifically, the module deals with the secondary power steering function that must take over in the case of a malfunction in the primary power steering function. Since the C-code itself is proprietary, we shall only describe its relevant aspects here, and will not be able to show any parts of it.

The code base of the analyzed module has 1,400 lines of code, consisting of 10 C-functions, one main and 9 helper functions. The analyzed code is strictly *sequential* (although the larger system is not), and the control flow consists solely of `if`- and `switch`-statements, and function calls (i.e., it does not involve any loops). The module interacts with two other modules: one primarily concerned with diagnostics, and one performing the I/O to the larger system. We had no access to the source code for the first of these, and could therefore not perform any reasoning about variables that depended on it. In the case of the latter, 9 small functions concerning reading, writing and status checking of signals were annotated as part of the verification. Additionally, the analyzed code makes much use of type definitions and macros imported from several external files, none of which were taken into account in our quantitative assessment.

Scania has its own internal programming rules for embedded systems, most of which are identical to the MISRA C development guidelines. Because of this, the analyzed code base avoids many of the C constructs which may cause problem in the stricter model which VCC operates in.

Requirements. Our starting point for annotating the code base was an internal document, containing 27 requirements. Of these, 14 were not considered for verification: 6 were not specific to the module itself (they had to do with initialization, and should be specified on another module), 3 were of a temporal nature (and thus could not be captured through VCC assertions), and the remaining 5 depended on output from other modules (and would need more modules to be included in the verification effort). Thus, 13 requirements were considered, of which 10 were verified due to time constraints. No functional errors were found in the code base during this verification.

The requirements are given in two formats: some are expressed in natural language only, and some in a semi-formal form, making use of logical statements and operators, such as `if`, `else`, `and`, and = (although the precise semantics of these operators is left unspecified). The document provides no details as to how variables referred to in requirements are related to system memory. After careful analysis we found that they could refer to globally available signals, local program variables, or not exist as explicit variables in the code at all.

An example of a semi-formal requirement was given above, in the beginning of Sect. 2. An example for a requirement in natural language could be: "*If the vehicle is moving without primary power steering, then the secondary circuit should handle power steering.*" The requirement can be seen formalized and

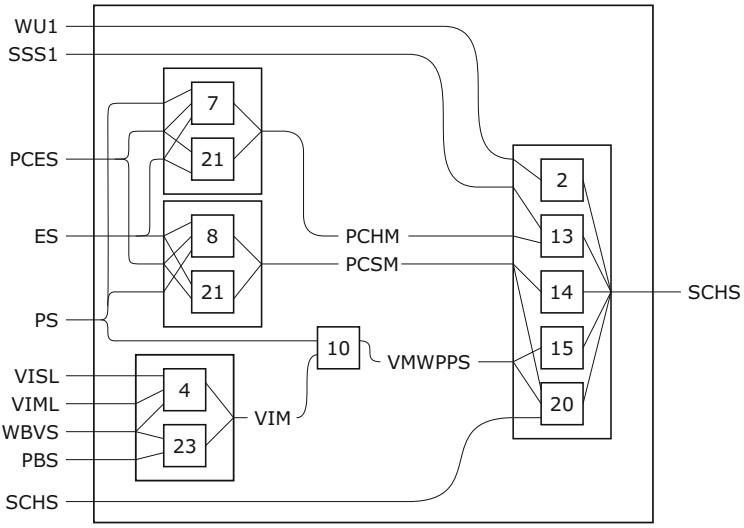


Fig. 3. Combinational logic circuit of case study requirements.

annotated in Fig. 5, and represented as a numbered box in Fig. 3, as requirement number 15.

A representation of some of the requirements in the form of a *combinational logic circuit* is shown in Fig. 3. The circuit models all requirements that define the value of the variable `SCHS` (the complete model is over 35 requirements variables, of which 6 output, 17 input and 12 intermediate variables). Outside the larger box are the interface variables, with input variables to the left and the output ones to the right. Requirements are represented by small boxes with numbers corresponding to the number of the requirement according to the document. These boxes are (sometimes cloned and) grouped together in "gates", so that each gate has a single output "wire" modelling a requirements variable.

The requirements did not form a complete specification of the module. We also found that under a naive interpretation two of the requirements were contradictory. Upon further investigation we found that this was a case of imprecise specification, and that they were intended to be evaluated in a certain order. Formal verification generally helps with detecting and resolving such issues, as two contradictory requirements can not both be verified on the same code base.

Code Preprocessing. Before verifying the requirements, the code had first to be prepared in order to pass VCC as it is, without any annotations concerning the application-specific requirements. First of all, some preprocessor directives concerning conditional inclusion of platform-specific headers and compiler-specific language extensions had to be either rewritten or removed. Furthermore, VCC always tries to prove validity and ownership of all accessed memory, which means that annotations for that purpose had to be inserted. We also chose to insert annotations for verification of termination of all functions before verifying the

requirements, which was easily achieved because of the simplicity of the module's control flow.

Code Annotation: Main Function. Since the analyzed module contains a single entry-point function, all requirements chosen for verification had to be specified in the contract of this particular function. The requirements were specified according to the white-box view described in Sect. 2, and were verified individually. To achieve this, ghost variables were used in the annotations to reason about the variables that do not exist as memory in the scope of the top-level function, such as local variables within a function, and about the requirements variables that are not implemented in the module. Variables of the former kind were referred to directly by their memory location. An example of a contract specified according to the white-box view is shown in Fig. 5. Some requirements were also specified according to the black-box view, in order to compare the readability of the resulting contracts. This was performed by substitution of model variables for the expressions which defined them.

Code Annotation: Ghost Variable Assignment. In order to successfully verify the white-box view contracts, the ghost variables have to be assigned the correct values during execution of the function. During the case study, we came up with two distinct methods to achieve this.

The first of these is to simply assign to the ghost variable the value of the local program variable or expression that it represents, within the functions where it changes. For manual annotation this is relatively straight-forward. The verification is also fast since the ghost representations are continuously synchronized with the actual code, and as such there is less work for the verification tool to prove the correlation. However, because of the tight connection to the code this method does not lend itself well to automation, since the relation between intermediary requirement variables and expressions in the code can not be inferred without human instruction.

In the second method, we define a separate ghost program that computes the complete combinational logic circuit discussed in Sect. 2, from program input variables. The ghost program can then be inserted in the body of the top-level function through inlining. Because VCC will need to infer the relation between the ghost circuit and the actual code, the performance of the verification is worse (how much worse depends largely on the number of intermediate variables, i.e., how much VCC needs to infer that would otherwise be explicit). On the other hand, the construction of such a ghost program from a formal requirements specification is far easier to automate. Another drawback of this method is that incomplete specifications cause problems, since VCC is not able to infer any relation between the ghost circuit and the software for input values that are not specified, whereas in the first method this relation can be made explicit even if not specified, provided ghost variables are always assigned values in the code.

```
int state[NUM_SIGNALS]; // Global state

int _(pure) read(int idx)
  _(requires \thread_local_array(state, NUM_SIGNALS))
  _(requires 0 <= idx && idx < NUM_SIGNALS)
  _(ensures  \result == state[idx])
{
  if (idx >= 0 && idx < NUM_SIGNALS)
    return state[idx];
}
```

Fig. 4. A fully specified function.

Code Annotation: Helper Functions. Since the requirements only specify the behaviour of the module as a whole and not how individual functions should behave, and because VCC performs its verification function-modularly, it was necessary to decompose and propagate the top-level requirements through the call hierarchy of the module. We utilized two complementary approaches to this, which we term *bottom-up* and *top-down*.

In the bottom-up approach, we give a complete specification of the computations performed by the functions, starting at the bottom of the call hierarchy, working upwards. This approach is suitable for small functions, which many other functions depend on, such as setters and getters, since giving a complete specification for these is relatively easy, and we get much value out of having one. An example of a fully specified, simple read function is given in Fig. 4.

In the top-down approach, on the other hand, we instead work with one requirement at a time, follow the trail of execution affecting that requirement through the functional hierarchy, and add the appropriate annotations, or partial specifications. This approach is suited for large and high-level (w.r.t. the call hierarchy) functions, where giving a complete specification is complex and not much value is gained from having one. An example of a partial contract of a high-level function, where annotations for only certain requirements have been supplied, is given in Fig. 5. These annotated requirements are represented as boxes within gates in Fig. 3, with their respective numbers.

Obstacles. Apart from the previously mentioned challenges with the requirements themselves, we also identified several obstacles to verification that could occur from how the code is written. Most importantly, the code should be written in a *type-safe* manner. To perform its reasoning, VCC must be able to lift the code to its own stronger model. If the code is not written in a well-typed manner, such as making use of implicit type conversions or aliasing of distinct memory objects, VCC will be unable to do this lifting and verification will not be possible without assistance in the form of additional annotations.

Another obstacle that may occur is code that depends on previous executions, for example in the form of local *static variables*. Such variables are outside the scope of the contract and can therefore not be used to directly specify

```
_(ghost \bool model_vehicleIsMoving) // Intermediate ghost variable
_(ghost \bool model_VehicleMovingWithoutPrimaryPowerSteering)

void steering()
  _(writes \array_range(state, NUM_SIGNALS))
  _(writes &model_vehicleIsMoving)
  _(writes &model_VehicleMovingWithoutPrimaryPowerSteering)
  // Req. 4
  _(ensures \old(state[WHEEL_BASED_SPEED]) > VEH_MOVING_LIMIT
        ==> model_vehicleIsMoving == \true)
  _(ensures \old(state[WHEEL_BASED_SPEED]) < VEH_STATIONARY_LIMIT
        ==> model_vehicleIsMoving == \false)
  // Req. 10
  _(ensures \old(state[POS_SENSOR]) == NO_FLOW
         && model_vehicleIsMoving == \true
        ==> model_VehicleMovingWithoutPrimaryPowerSteering == \true)
  // Req. 15
  _(ensures model_VehicleMovingWithoutPrimaryPowerSteering == \true
        ==> state[SECONDARY_CIRCUIT_HANDLES_STEERING] == \true)
```

Fig. 5. A partially specified function.

properties of the function; but at the same time changes to their values may affect future invocations of the function. It is possible to work around this, for example by connecting static variables to ghost variables that exist in the scope of the contract, but a much simpler solution is just to try to avoid them.

Variables of the enumerable and Boolean types can also make verification difficult in some cases, since in the C language they are in reality backed by the integer type, and may assume all the same values. While these types of variables are common and may not be easily avoided, they introduce some additional annotation effort; for a successful verification of requirements referring to such variables, annotations for proving that the variables never assume values outside of their expected domain are usually needed.

5 Discussion

Summary of Findings. We now return to the questions raised in the Introduction.

1. We found the *formalization* of functional requirements intuitive to achieve, and without much effort. We also found that the formalization helped clarify the requirements, as we were forced to resolve ambiguities and contradictions in order to achieve a valid verification.
2. The *code coverage* of the verification tool VCC for the given code base was almost complete. It is relatively easy to use the tool, especially as it can be configured as a plug-in to Visual Studio, but requires a certain training and knowledge of the underlying verification technology to make full use of it.

VCC turns out to be relatively *efficient*: it took 165 sec to verify the whole annotated module, of which 65 sec went to the "worst" function. On the negative side, the *feedback* provided from the tool when verification fails only highlights the specific assertions that failed to verify, without any hints as to why. Depending on the type and complexity of the assertion, this feedback may not always be useful, and careful analysis of the code and the annotations is usually required to understand what went wrong.

3. The *annotation overhead* of the code was about 50%, or roughly 700 lines of annotations. The annotation was performed manually, but we observe a clear potential for automation of (most of) the annotation process (see below). Manual annotation of the code, even after having formalized the requirements and understood the tool and code base, required much effort; we estimate it roughly to have taken between 1 and 1.5 person-months. In particular, finding and inserting appropriate annotations for all memory accesses, as well as figuring out how each function affects the individual requirements or, alternatively, giving a full specification for the function, are time consuming tasks.

Towards Semi-automated Specification and Verification. Based on findings from the case study, we propose the following work process that automates most of the specification and verification effort, as a prerequisite of integrating our technology into the development process for safety-critical embedded C-code.

Our proposal is to start from a (potentially graphical) combinational logic circuit-like description of the computed functions, according to a chosen function decomposition (i.e., a white-box view as illustrated on Fig. 3), together with descriptions of the individual "gates" of this circuit, created with the help of a tool. The tool has to support specifying interface requirements variables in terms of references to the actual global program variables, or otherwise allow this mapping to be provided by the user separately. This description can be seen as the *requirements model*, and is then to be translated to a VCC "ghost program" computing the functions, by introducing a ghost variable for each requirements variable. In this way we can utilize the existing syntax and operational semantics of VCC ghost code, and are thus relieved from the need to have to define such a semantics for a new formal requirements language. This ghost program can thus be seen as an *executable specification*.

From the requirements model, including the mapping of interface requirements variables to actual global program variables, a *contract* for the main function of the given module is to be generated. The tool should support generating both the white-box and black-box view contracts. In the white-box view, global program variables should be used in the specification of interface requirements, in order to enable modular verification. The generated executable specification is to be inlined in the top-level function of the module, so that the value of the intermediate requirements variables can be computed. As a fallback strategy, intermediate requirements variables may instead be manually synchronized with their program counterparts, in cases where verification proves unfeasible.

What then remains to be annotated are the helper functions. One way of handling these is to *inline* them successively into the main function. While this eliminates the need for annotation of helper functions altogether, its drawbacks are the potential explosion of code (which may result in an inability of VCC to verify it), and the need to maintain a verified code base separately from the actual code base, creating a potential gap and making more difficult the interpretation of the feedback from the tool.

The preprocessing phase described in Sect. 4 is to be assisted by dedicated static analyses, which are to generate annotations for the different types of implicit requirements. Certain postprocessing may also be needed in order to help the programmer in making sense of and reacting to the messages that VCC issues on unsuccessful verifications.

6 Related Work

VCC has been used in a number of software verification initiatives. It was in fact built with verification of the Microsoft Hyper-V hypervisor in mind [5,7]. In a case study [2], VCC was also used to verify another hypervisor, although a less complex one. The study presents techniques for verification using automated methods. It describes modeling of interaction between hardware and software, and shows that functional verification of simulation of guest machines is feasible. In another case study [3], VCC is used to verify system calls in a micro-kernel based operating system targeted at safety-critical embedded systems. The study was part of an avionics project, and describes the verification process as well as how the underlying hardware architecture was modelled. In addition, it is shown that assembly code can be semantically specified and integrated in the verification through VCC.

Within the same avionics project, a case study utilizing the verification tool Frama-C was also performed [9]. The study evaluates several aspects of modern formal verification, such as how formalization of requirements can be achieved and when it is feasible, and the complexity of the formal languages of verification tools in comparison to programming languages. Solutions to many obstacles that commonly occur in formal verification are proposed. Of the encountered case studies, this is the only one with a starting point similar to ours, i.e. informal requirements specifying functional relations between input and output states. Our approach is different in that we formalize the requirements as a circuit, which can then be executed in ghost code, as well as handle requirements variables without explicit counter-parts in the software.

A methodology for reasoning about timed and hybrid systems in VCC is presented in [4]. The approach uses what is referred to as *Timers* and *Deadlines*, and can provide a solution to the verification of temporal requirements in a functional setting. Another work examines the incorporation of strongest postconditions in the verification process, and how symbolic execution can be used to calculate them [11]. Such a framework could provide a basis for automation of much of the C code annotation process, particularly the (complete) specification of function contracts.

7 Conclusion

In this paper we summarize our findings and experiences with specifying and verifying deductively the functional requirements of an embedded safety-critical C-code module, by using the VCC tool. The main specifics of the verified code is that it computes a multi-output function over variables from finite domains that has a non-trivial, multi-level decomposition. The main challenge then is how to deal with intermediate requirements variables.

The pre-study indicates that deductive functional verification can be a viable option for increased quality assurance of safety-critical embedded C-code. For its integration into an *embedded C-code development process*, however, a number of issues need to be resolved. First, a formal requirements language needs to be adopted and guidelines for writing requirements need to be formulated and supported by a tool. Second, the coding rules that are prerequisite for successful verification need to be enforced. Third, the annotation process needs to be automated almost completely, with clear hints to the programmer where he or she has to provide annotations, and of what type. And fourth, support for interpreting and handling the feedback from the verification tool needs to be provided in a way that allows unsuccessful verifications to be resolved adequately and without requiring deep knowledge of the inner workings of the tool. Our work currently focuses on addressing these issues.

References

1. Allocation Element Requirement AE417 Dual-Circuit Steering. Scania Technical Product Data (2015)
2. Alkassar, E., Hillebrand, M.A., Paul, W., Petrova, E.: Automated verification of a small hypervisor. In: Leavens, G.T., O'Hearn, P., Rajamani, S.K. (eds.) VSTTE 2010. LNCS, vol. 6217, pp. 40–54. Springer, Heidelberg (2010). doi:10.1007/978-3-642-15057-9_3
3. Baumann, C., Beckert, B., Blasum, H., Bormer, T.: Formal verification of a microkernel used in dependable software systems. In: Buth, B., Rabe, G., Seyfarth, T. (eds.) SAFECOMP 2009. LNCS, vol. 5775, pp. 187–200. Springer, Heidelberg (2009). doi:10.1007/978-3-642-04468-7_16
4. Cohen, E.: Modular verification of hybrid system code with VCC. CoRR abs/1403.3611 (2014)
5. Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: a practical system for verifying concurrent C. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 23–42. Springer, Heidelberg (2009). doi:10.1007/978-3-642-03359-9_2
6. Cohen, E., Moskal, M., Schulte, W., Tobies, S.: A practical verification methodology for concurrent programs. Technical report MSR-TR-2009-15, Microsoft Research, February 2009
7. Cohen, E., Moskal, M., Schulte, W., Tobies, S.: A precise yet efficient memory model for C. In: Workshop on Systems Software Verification (SSV 2009). Electronic Notes in Theoretical Computer Science, vol. 254, pp. 85–103. Elsevier (2009)

8. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C. In: Eleftherakis, G., Hinchey, M., Holcombe, M. (eds.) SEFM 2012. LNCS, vol. 7504, pp. 233–247. Springer, Heidelberg (2012). doi:10.1007/978-3-642-33826-7_16
9. Dordowsky, F.: An experimental study using ACSL and Frama-C to formulate and verify low-level requirements from a DO-178C compliant avionics project. In: Formal Integrated Development Environment (F-IDE 2015), pp. 28–41 (2015)
10. Eriksson, J.: Formal Requirement Models for Automotive Embedded Systems. Master's thesis, KTH Royal Institute of Technology, School of Computer Science and Communication (2016)
11. Gordon, M., Collavizza, H.: Forward with Hoare. In: Roscoe, A., Jones, C., Wood, K. (eds) Reflections on the Work of C.A.R. Hoare, pp. 101–121. Springer, London (2010). doi:10.1007/978-1-84882-912-1_5
12. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969)
13. Lidström, C.: Verification of Functional Requirements of Embedded Automotive C Code. Master's thesis, KTH Royal Institute of Technology, School of Computer Science and Communication (2016)