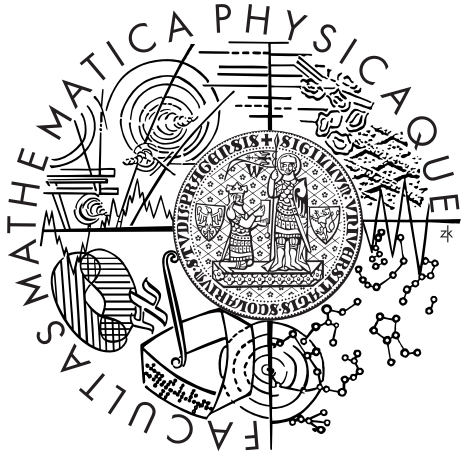
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Bc. David Škorvaga

Analysis of a File System Using the Verifying C Compiler

Department of Distributed and Dependable Systems

Supervisor of the master thesis: RNDr. Jan Kofroň, Ph.D.
Study programme: Computer science
Specialization: Software systems

Prague 2014

I wish to thank my advisor for his valuable advice, guidance and enough patience during consultations of the thesis.

I would also like to thank RNDr. Ondřej Šerý, Ph.D. for his advice and assistance with VCC tool.

My special thanks go to my family for their support during my studies.

I dedicate my master thesis to Libuše, my mother.

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In Prague, December 5, 2014 David Škorvaga

Název práce: Analýza souborového systému pomocí Verifying C Compiler

Autor: Bc. David Škorvaga

Katedra: Katedra distribuovaných a spolehlivých systémů

Vedoucí diplomové práce: RNDr. Jan Kofroň, Ph.D.

Abstrakt: Formální verifikace je jeden ze způsobů, jak zlepšit spolehlivost softwarových systémů. Jeden z přístupů formální verifikace se zaměřuje na dokazovaní správnosti anotovaného zdrojového kódu v široce používaném programovacím jazyce. Verifier C Compiler (VCC) je verifikátor pro concurrent C, který přijímá anotovaný kód v jazyce C a automaticky ověřuje jeho správnost s ohledem na tuto anotaci. Už se objevily úspěšné pokusy o ověření některých kritických systémů, včetně jádra operačního systému. Další důležitou součástí operačního systému je jeho systém souborů. V diplomové práci jsme si vybrali souborový systém FatFs, odlehčenou implementaci souborového systému FAT, nezávislou na zařízení. V této práci vytvoříme specifikaci jeho části pomocí anotace VCC a úspěšně ověříme jeho korektnost.

Klíčová slova: Formal Verification, File System, VCC

Title: Analysis of a File System Using the Verifying C Compiler

Author: Bc. David Škorvaga

Department: Department of Distributed and Dependable Systems

Supervisor: RNDr. Jan Kofroň, Ph.D.

Abstract: Formal verification is a way to improve reliability of software systems. One approach of formal verification is focused on proving correctness of annotated source code of an established programming language. Verifying C Compiler (VCC) is a verifier for concurrent C that accepts an annotated code in C language and automatically verifies its correctness with respect to the given annotation. There have been successful attempts to verify some critical systems, including the operating system kernel. Another critical part of operating system is its file system. In the thesis, we choose FatFs file system, a simple device-independent implementation of the FAT file system. We specify a part of it using the VCC annotation and successfully verify its correctness.

Keywords: Formal Verification, File System, VCC

Contents

Introduction

Formal verification (or program verification) is one of the most effective methods of software analysis. It may improve software reliability and if we assume the correctness of verifier and hardware, it is the way to guarantee that a system is without errors. Static analysis and formal methods are used particularly in creating mission-critical, safety-critical, cost-intensive and embedded systems that have a goal to reach some level of software quality. The increase of usage in commercial sphere is caused by a need of verifying properties of programs used in computer systems which are vulnerable in some point of view of computer security or which control machines that cannot fail.

The main aim for use of formal methods is to detect flaws in product requirements, design and implementation, not only concrete bugs in already written programs. There are areas that are highly sensitive to any failures in the entire system. There is an enormous increase of using personal electronics with embedded systems for everyday use, such as smart mobile phones, mobile computers and household appliances. But not only them, we may also mention industry branches as intensive care, medical machines and control systems, electricity transmission and distribution, robotics, nuclear engineering, anti-missiles systems and telecommunication. A significant application of these methods is in transportation, e.g. aviation industry and air traffic control information systems, railway signaling systems, spaceship program and automotive industry. Equally important is financial sector and security, e.g. access control, electronic banking, credit and phone card security, cryptographic protocols and electronic signature, etc.

As we can see, formal approach has more and more justified in application development. All of these areas are using safety-critical systems whose breakdown may cause serious injury to people or extreme damages of property or equipment and therefore the use of formal methods is very likely to be cost-effective. For this purpose, corporations and standards organizations publish software certificates and technical standards, e.g. DO-178B by RTCA, IEC 61508, ITSEC E6, TCSEC A1 and ARINC standards. For instance, standards ITSEC E6 and TCSEC A1 require, among other, formal specification of declared system properties with a proof of correctness.

As the complexity of software increases, the manual verification becomes more difficult and time-consuming. Progress in the development of advanced deductive methods allow to create tools that carry out semi-automatic or almost automatic verification. There has already appeared achievements in formal verification of real-life systems such as NICTA's Secure Embedded L4 microkernel (seL4)[18] that is the first verified general-purpose operating system kernel, or SYSGO's PikeOS[3] that made use of Concurrent C Verifier (VCC) created by Microsoft research. We employ exactly this tool to verify the program that can be another challenge after OS kernel: a file system[16].

The goal of the thesis it to choose a proper existing real file system, create a specification for its source code using the VCC annotation and analyze it with the VCC verifier. The file system should be simple enough to make verification even possible, but on the other hand, robust and preferably used in practice. The FAT file system may be a good choice. We have chosen open source FAT

driver called FatFs for this purpose. In the work, we also document provided specification, found bugs and code source changes and justify what has or has not been possible to verify.

The first chapter describes a background of the thesis. We introduced what is formal verification, correctness of program, formal methods and semantics of programming languages that are extensively used by automatic verifiers.

The second chapter contains a brief manual of VCC tool. Since the VCC is complex verifier, we mention only constructs used in the file system annotation. There are described basics like assertions, preconditions, postconditions, class and loop invariants. There is provided also explanation of more complicated mechanisms like type system, ownership, claims, etc.

In the third chapter, we briefly show the architecture of FAT file system that is the underlying file system for FatFs module. Then, we describe the FatFs itself. It defines public interfaces, objects and private static functions that we annotate and verify in the next chapter.

The fourth chapter is the documentation of the practical part of the thesis. We document mainly the specification of public objects and some important details that have appeared during the verification of static private functions. We also present all necessary modifications of the FatFs implementation (including fixed bugs).

1. Formal verification

Formal verification (or generally program verification) is a process that check correctness of an algorithm with respect to the formal specification, using formal methods or mathematical proving [27]. The verification is done by providing a formal proof in an abstract mathematical model of the program, so it is essential to formalize the entire system and represent it as a mathematical model. In the area of computer science, formal verification applies to both hardware and software systems because both of them can be represented as a mathematical model. Providing of complete formal proof of correctness for real-world hardware or software is difficult.

It must be proven that formal system correspond to expressiveness of the system that is used for development of program. There are many models able to be suitable for representing algorithms, used both in computability theory, e.g. lambda calculus and process algebra, as well as models that respect syntax and semantic of programming languages and they are suitable for reasoning rigorously about them, e.g. axiomatic semantics and Hoare logic with separation logic.

Verification presents a different approach to checking program correctness than software testing, especially dynamic testing including unit testing that is one of the most commonly practiced form of testing. Common testing methods are well scalable, easy to create and also inspect whether the program meets the specification. However, software systems are nowadays complex and still bigger and grow faster. Application development is getting cheaper, but testing is getting still more expensive and does not keep pace with the development.

Moreover, even for small systems, the simulation or testing of behavior at runtime can hardly to be exhaustive and impossible in the case where the software may consume unlimited amount of data. Testing is only able to demonstrate that the program is incorrect, no matter how many tests can successfully pass. Unlike verification that, rather than relying on the abilities and perfection of designers and programmers, has the potential to prove that the program is correct. We may classify formal verification as more effective part of static software analysis, other than syntactic methods or bug finding.

In general, it is neither suitable nor feasible to fully specify and afterwards verify large software systems. It is almost impossible to show correctness of entire large systems. The main point of formal verification is not to replace testing. The usual approach of formal verification is to prove some properties and fulfill some requirements on software. Formal methods are applied to the most critical part or modules in software system. We can mention some advantages of formal verification:

- Formal proof can replace most of testing because it can cover most of possible input data.
- Formal methods are well-suited for automatic reasoning and may be used in automatic test cases.
- To perform the verification, we must precisely specify requirements on the system. It may improve quality of specifications and also result in better documentation.

Upon verification, we can consider proceeding in two ways. When new software is created from scratch, it is highly desirable to verify the program in parallel with its development. This will help to improve both implementation and specification that may not be so detailed in the early stages if development. However, most of software is implemented at first and if it proves to be benefit, it is verified additionally. Then, the verification is naturally much harder. Even more difficult is the case when there is no quality specifications or even software documentation.

As programs grow, it is becoming harder to create a mathematical proof of them by hand. Therefore, there arise and promote tools for the automatic analysis more often. It is valuable approach for many reasons: It reduces testing and quality control of the software and does not demand such claims to knowledge of developers. It also prevents human error when creating the proof, if the automatic verifier is correct. In that case, verifier must be also correct. These tools may be applied to sequential or concurrent software which are more valuable because of increasingly common multi-processor platforms and because writing of multi-threaded programs is hard.

1.1 Correctness

If we talk about correct programs, we should explain more precisely what correctness exactly is. We claim that algorithm is *correct* if it fulfills its specification [4]. If program does what is declared in the specification, then we can talk about *correctness* of the algorithm. Correctness depends on the type of problem we are trying to solve and thus we cannot simply say that the program is correct without stating what the program should do. To show that an algorithm meets its specification, we must provide a formal proof of correctness, assuming that both the algorithm and specification are written formally.

The most common type is *functional correctness* that requires an algorithm to produce correct computational result. We define functionality of the system in functional requirement where the function of the system is described as a list of all possible inputs and subsequent outputs together with appropriate behavior. Then the algorithm is correct if for every input it produces the proper output written in the requirement. Another type is timing correctness requiring that program computation must finish within a predefined time period. It is an elementary attribute of real-time operating systems.

Based on how the correctness is strong, we may distinguish total and partial correctness [4]. *Partial correctness* of an algorithm means that it is only required that if algorithm match its specification and the execution of program terminates than the algorithm is correct. On the other hand, *total correctness* is stronger than partial one and an algorithm is totally correct if it is partially correct and execution of the program always terminates. These two conditions are usually proved separately.

Since it is known that Halting problem is undecidable, we cannot say for every program whether it terminates or not. Consequently, program verification is undecidable. However, if we restrict the application domain, we are able to verify some interesting properties in most usual programs. There exists an analytical technique described later that can say that computation of a program will definitely terminate. From the perspective of programming languages, most

of elementary constructions always finish, only few are problematic, such as **while** loop, recursion and blocking subroutine calls.

Correctness of the algorithm itself does not guarantee that the program can not fail. Here we can distinguish some possible reasons for an error in the program:

- Algorithm is not correct. It means that it does not satisfy the specification. During verification, we have found an error in program and we should fix its implementation. This kind of error is called malfunction.
- Program satisfy the specification, but it is not valid. It indicates that there is a inconsistency in specification itself and we should fix it together with program implementation. Formal verification cannot often detect these errors because since both specification and algorithm are wrong, verification may succeed.
- Program has failed because of reason that is not mentioned in specification. Even if the algorithm is correct, it does not mean that its implementation in the given machine cannot fail. It may always happen something that we do not expect or we expect it, but we do not care. Classical example could be limitation on computer memory.
- Error is in the environment such as compiler, operating system, hardware. We cannot generally avoid these types of error, unless we know that the environment is verified too.

We must be careful when we talk about valid programs. Both terms are closely related, but validity of the program is different term than correctness. We can say that program is valid if it works as we expect. Then, validation is a check that developed software meets the needs of a software user. For this reason, validation admits only dynamic testing against some test cases because inspection of source code is not interesting for the user.

1.2 Formal methods

To achieve formally defined and verified system, we should employ proper methods. *Formal methods* [22] is[1] an approach to proceed in creating of the entire system. It is suite of engineering and mathematically based techniques whose goal is to perform appropriate formal analysis to refinement of the system and contribute to improve its reliability and quality. Formal methods may be applied to both hardware and software part of the system and its point is to formalize individual parts of application, including formal specification, implementation and subsequent formal verification. We focus on formal methods used during verification.

Formal methods used in software developing prove properties from programming language itself. It has its semantics and it is necessary to formalize it. There exist three main groups of formal semantics: denotational, operational and axiomatic semantics [28]. In this text, we are interested in proving *axiomatic*

[1] The phrase "formal methods" has two meanings: area of practice and set of various methods. in the first case, it is noun phrase where verb is used in the singular.

semantics. This semantics is more close to the term of command in programming language. Every command has an effect on assertions about the program state. Then we can recognize initial assertion before the command execution and final assertion placed after it. Assertion has form of predicates containing variables which define the current state of program. The relation between these two assertion capture the semantics of the language. As the program runs, the values of individual variables are changing, but the relationship between them is maintained. The best known example of this semantics is Hoare logic, described later.

Originally, the formal proof was made by hand, but as system grows, it was necessary to automate the process. There were created partly or fully automated tools. Semi-automated (or human-directed) tools only suggest next steps and are not able to prove more complicated assertion and user interaction is necessary, but the proof can be effectively checked by them. However, the main goal is a fully automated verification tool. It is hard in general, though there are common practices to achieve it, e.g. to apply the method only on the high-level design without most of details, divide system hierarchically to separate modules, restrict some complicated details as ranges of variables etc.

There are basically two branches of automated verification techniques: model checking and deductive verification [26, 24]. *Model checking* is a technique for verifying finite state space of sequential, but mainly concurrent programs where it may traverse all interleavings. It systematically explores all possible behaviors and check if program meets specification. It is fully automatic and usually fairly fast. Moreover, if it finds an error, it produces a counter-example. However, number of states grows fast with respect to the size of data and the number of threads (state explosion), if it is not given a proper abstract model, hence it is not suitable for larger programs. Temporal logic is usual the language of specification.

We are interested primarily in deductive verification, especially automatic theorem proving. Deductive verification uses symbolic representation and thou does not suffer as much from the state explosion. Theorem provers work with specifications written in some expressive logic, often in a first-order logic, but may be also in a higher-order logic, which is strong enough to prove useful assertions.

The aim of these tools, as it is evident from the name, is to find a proof of correctness as an output. This proof is usually much longer than it is necessary, as prover makes progress mechanically. Moreover, if the deductive method fails to find the proof, it does not produce a counter-example. Another disadvantage is that theorem provers are usually not fully automatic, in the sense that they cannot make decision only from specification, and thus they require a skilled user that controls the process. Some tools demand an interaction with the user, others just need suggest intermediate steps that they should pass when proving.

1.3 Hoare logic

Hoare logic [14] is a formal system with a set of logical rules for reasoning rigorously about the correctness of computer programs. In this mathematical model, a programming language is viewed with axiomatic semantics. It introduces valid axioms and inference rules that are closely related to elementary commands and statements of a programming language, thus these rules are different for every

language. The logic was presented by British computer scientist C. A. R. Hoare.

The Hoare logic works with concept of program state. Every command in programming language is defined such that it changes the program state. We may claim various predicates about the state and we call them *assertions*. These assertion are formulae in a first-order logic (or in predicate logic in general). From the view of a command, there are two states, the state before the execution of the command and after it. In Hoare logic, the principle construct is so-called *Hoare triple*:

$$\{P\}\, C\, \{Q\}$$

where P and Q are assertions and C is a command. This notation means that the command may be characterized by assertions at the entry of command, called *precondition*, and at the end of command, called *postcondition*. In original Hoare logic, Hoare triple shows only partial correctness and we can interpret it as follows: If the execution of command C starts in a state satisfying precondition P and terminates, then it results in a state satisfying postcondition Q. In the case of total correctness, the Hoare triple shows that the command C always terminates and we usually write it as $[P]\, C\, [Q]$, although program termination is usually proved separately.

1.3.1 Loop invariants

Most of useful algorithms use loops. All axioms and rules of classic Hoare logic are sound for partial correctness as well as total correctness, apart from the **while** loop, because it may never terminate. Hoare logic presents **while**-rule for simple **while** loop:

$$\frac{\{P \wedge S\}\, C\, \{P\}}{\{P\}\, \text{WHILE } S \text{ DO } C\, \{P \wedge \neg S\}}$$

where assertion P is called *loop invariant*, S is the loop condition and command C is loop body. Now we explain the meaning of the rule: If $\{P \wedge S\}\, C\, \{P\}$ is true, then P is invariant of C whenever S holds.

The **while** rule says that if P is an invariant of the body of a **while** command whenever the test condition holds, then P is an invariant of the whole **while** command. It is important for reasoning because it does not matter how many times C executes

If we want to prove loop invariant, we must show three things:

- The invariant holds at loop entry, in the state before the loop.
- The invariant holds after executing loop body. All values in variables that may be changed in the loop are unknown and only invariant and loop condition is guaranteed at the beginning of loop body.
- The loop eventually terminates. We must find variant of the loop and prove that decreases to ensure termination.

1.3.2 Pointers and local reasoning

Things start to get complicated when we think about pointers and the heap. We can prove programs that manipulate with pointers in Hoare logic, but it is clumsy in the original version. It is difficult to represent the heap with basic axioms of Hoare logic and it would be much easier to introduce additional axioms that would reflect the true nature of pointers, as they used in imperative programming languages. Exactly this is done in *Separation logic*, the extension of Hoare logic.

In the Hoare logic, the state is represented as a function from variables to values. Separation logic extends the state pair where the first element is the original mapping from variables into values, and the second element is mapping from memory addresses to values. It is evident that there is only a partial function from addresses to values because the heap is allocated dynamically. Note that addresses may be represented as numbers which is the case of the C language memory model. Languages such as C# or Java do not allow manipulation with pointers, thus they may represent addresses as tokens. There exists the null pointer which indicates that the address is always invalid.

Separation logic introduces a group of new axioms and inference rules that specify the behavior of four new operations: Fetch and heap assignments that add ability to write and read values in heap, and allocation resp. disposal operation that validates resp. invalidates memory addresses. The manipulation may evolve into two fundamental problems:

- Access to value of address that is invalid. Separation logic defines this situation as a fault. It extends the meaning such that now if precondition is satisfied, the execution of command must not fail. Standard axioms are modified for that reason.

- More than one variable keeps the same address. This is called aliasing and it is a real problem for local reasoning. We need to ensure that when we execute a command, all conditions not affected by it are not changed, i.e. when command access a memory footprint of the heap, other memory is not changed. For this purpose, separation logic introduces *Frame rule*:

$$\frac{\{P\}\, C\, \{Q\}}{\{P * R\}\, C\, \{Q * R\}}, mod\,(C) \cap fv\,(R) = \emptyset$$

  where $mod\,(C)$ denotes the set of modified variables in command C, $fv\,(R)$ are all free variables in condition R and $*$-operation is variant of separating conjunction where both conditions hold in two disjoint parts of the heap. It simply says that command executed in a footprint and satisfying postcondition, can also execute in an extended footprint and the additional memory is not changed by the command.

Features added by separation logic are useful in some common situation:

- Modular development where program parts (functions, modules etc.) access separate memory and do not affect other parts.

- Manipulation with pointers, especially in data structures as lists, trees etc.

- Transferring of ownership that identifies the owner that has the responsibility for the memory: it has permission to access the memory (concurrent programming) or has obligation to deallocate the memory.

2. Concurrent C Verifier

Verified Concurrent C[8] (*VCC*, previously also called Verifying C Compiler) is a fully-automated industrial-strength verification environment and deductive verifier for concurrent C code. It performs contract-based modular analysis and verifies partial correctness of every function in annotated C code. It has been developed by Microsoft Research and the European Microsoft Innovation Center (EIMC) in Aachen as part of Verisoft XT [31]. The primary goal of the VCC was to verify Microsoft Hyper-V hypervisor software.

VCC defines its own specification language in the form of classical first-order predicate logic. It has C-like syntax and is written as arguments of the special function macro. It allows programmers to put specification as a code annotation directly inserted into source code and help them maintain the code according to their specification. Since the contract is macro, the annotation does not vitiate the program because the annotated code can be simply omitted by using conditional compilation.

VCC uses the deductive verification paradigm. It takes an annotated C program as an input. The annotated code may contain assertions, function preconditions and post-conditions (contracts), class invariants, loop invariants and ghost code with ghost objects. Then, it attempts to verify that the code respects the annotation (meets the specification), and prove the correctness of program, i.e. the specification holds for every possible program execution.

The tool does not verify the program by itself, but exploits two other tools. It translates the annotated C code into *Boogie*, the intermediate verification language. The Boogie tool generates a number of mathematical statements called verification conditions. If these conditions are valid, it only guarantees partial correctness of the program, total correctness is achieved by running of termination analysis (they may run together). The conditions are then put into *Z3*, an automatic theorem proper that tries to prove them. It either solves it, or does not terminate, or gives a counterexample. If a counterexample is found, VCC reflects the failure back into the source code itself, i.e. it reports that it cannot verify the correctness of corresponding annotations. The programmer does not see output from Z3 by default.

The verifier has some features that make it really powerful tool:

- **Soundness** – If VCC verifies a program, it is correct indeed, unless VCC would contain bugs. A sound system does not need to prevent false positives, but always prevent false negatives. It aims at really deep verification, not looking for bugs. If VCC asserts that a program is correct, it always runs without error and additionally meets the specification.

- **Modularity** – Analyzing a large software at once might be hard. VCC verifies functions and types of program one by one. If the static modular analysis is performed on a function, this caller function needs to know only the contract of the callees and invariants of all types it uses. So we may verify a program even if we have only function prototypes without bodies. This approach is essential to achieve good scalability of verification.

- Support for **concurrency** – Contract defines behavior of function if it is executed concurrently with another one. Function verification implicitly guarantees thread safety in any concurrent environment, but it has mechanisms that can work with shared objects. VCC may verify programs that use both coarse-grained and fine-grained concurrency and it may even verify synchronization primitives.

- Supports for **low-level C** – VCC was primarily used for verification of hypervisor that contains very low-level code. These features include bit-fields, unions, wrap-around arithmetic, etc.

In the next sections, we describe some basics about commands, structures and mechanisms used in VCC annotation. All the annotations are then used in verified source code. The text does not include how the tool works inside, but how it is used. For more detailed information, we recommend read the tutorial [30] and manual [29].

2.1 Basics

We begin with very simple program 2.1 that does not contain any contract or function calls. The program just performs some operation on uninitialized values. It does not access any memory thought a pointer and it has only one annotation inside its body.

```
#include <vcc.h>

int main(void) {
    int x, y, z;

    if (x >= 0) y = -x - 1;
    if (y >= 0) z = -y - 1;
    if (z >= 0) x = -z - 1;
    _(assert (x < 0 || y < 0) && (y < 0 || z < 0)
        && (z < 0 || x < 0))

    return 0;
}
```

Source code 2.1: Simple program with VCC annotation

As we can see, every program must include `vcc.h` header file. It defines preprocessor macro `_(...)`, where all VCC annotations must be placed. If a regular compiler builds a program, the macro is disabled and it is resolved to whitespace. The parameters of macro always begin with a command keyword, called annotation tag, which specify the purpose of the annotation.

Our program contains only one annotation in the form `_(assert E)`, called an assertion, where assert is a tag and E is an expression. If VCC decodes this command, it tries to prove whether the expression holds (it is evaluated to nonzero) in the current point of the program (it does in our example). If the verifier succeeds, it means that the expression holds in every execution of the program. This is contrary to assert function in C language that works only at runtime and even if succeeds, it does not mean that will succeed again it the next execution.

VCC also allows to make assumption in the form of _(assume E) annotation, which is, in a certain sense, counterpart to assert. If E does not hold, VCC ignores everything after the assumption. In other words, VCC considers the expression valid and may use it for the further reasoning. Ideally, all assumptions should be eliminated, but it is often not possible or desirable.

Sometimes we need to assume a fact that can be very hard or impossible to prove for VCC. It may have some reasons. VCC may not be able to prove such type of problem or the reasoning is so slow that the assumption is the only wise solution. Also, auxiliary assumption are very useful in the situations where we need to focus on verification of other properties than the assumed one.

Note that inside the assert and assume annotations, there cannot be inserted any expression that would change the context of the program. For instance, _(assert x++) is not allowed, because the program with annotation would behave differently than program without annotation. Certain annotations create a *pure context* where changes of program state are not permitted. Not only arithmetic or predefined expressions are pure. We may declare a function as pure if does not change any context, only local variables, with _(pure) cast.

2.2 Semantics

VCC defines a simple abstraction. From its point of view, everything is an object. The main property is that all these objects are independent (it is legacy of Spec#). Every object is uniquely defined by its type and its address and everyone has also a set of fields. These fields are also independent of each other and everyone has its type and dedicated value of that type.

Every object is either concrete or ghost. Concrete objects correspond to the data present in a running program (as an instance of **struct** data type). Concrete fields of every object have fixed size and offset and any two fields do not overlap. It corresponds to the real program. Ghost objects are data that are used only during the verification, i.e. they are not part of the C program. There is also ghost code that is "running" only during the verification, i.e. VCC makes reasoning over it. Ghost code cannot change concrete data and vice versa. Together, ghost data and code are the essential part of specification language and they are heavily used in VCC methodology. Ghost code is always defined in _(ghost ...) annotation.

Every concrete or ghost object may define any number of ghost fields. There are already several special ghost fields defined in every object that are precisely described in appropriate sections. The Boolean field \valid indicates whether the object is active. As we said, all objects are independent, but we may have two objects that have different types, but partly share one address space. In C language, simple inheritance may be implemented like this, but it is tricky solution, so VCC completely forbids address space sharing. Since concrete fields of valid objects don't overlap and a function accesses only valid objects, we may define an injective function from program fields into a flat shared C heap, so the verification simulates legal program execution.

Now we may define state of the program as mapping from all valid objects and field names to values. Every statement of the language moves from one state (prestate) to another state (poststate). The pair of states is called transition.

Every program execution is a sequence of states and transitions between them. Finally, in short, VCC successfully verifies the program if for every program execution, the first state holds all tested predicates (called invariants) defined both by VCC itself or C code, and, after every transition, invariants still hold. The semantics is based on Hoare logic.

2.3 Expressions

The expressions that can be created in C language, does not cover all the capabilities of first-order logic. Especially unlimited numeric types and quantifiers are missing. VCC introduces some additional elements that can be used in expressions added into annotation.

2.3.1 Integral types

There are three new integral types:

- \bool – Simple Boolean type. It can have only values \true and \false, which are equivalent to the values 1 and 0.
- \integer – Mathematical (unbounded) integers, i.e., arbitrary precision integers. Every integer may be represented by this type. C integral types can be cast to \integer.
- \natural – Unbounded natural number, subset of \integer[1]. C unsigned integral types can be cast to \natural.

By default all C integer types are mapped to mathematical integers. When we have an integral expression in annotation, it is automatically converted into the \integer type. Also, all operations with integers are extended to use \integer and \natural with standard arithmetic and even the result has this type.

However, not every pure expression is valid in concrete context. The problem is that the result of expression or even its sub-expression has no upper or lower limit and may overflow or underflow. Such expression cannot be either valid (represented in fixed-memory place) or saved into a variable of smaller type without casting. It may indicate a program flaw or that we just missed to add a condition (precondition or invariant) that prevents it. For natural expressions, it is usually wise to limit it from above in preconditions or invariants, e.g. a + b < (DWORD)−1.

The possibility of overflow may appear only in checked context. Conversely, some algorithms assume that the variable might overflow. It is well defined for unsigned integers. We may switch into unchecked context where all operations in expression may overflow. It is done by the operator _(unchecked). The expression then behaves exactly like expression in C language, e.g. _(unchecked)(x+y)== (UINT_MAX − x < y)? (y − (UINT_MAX − x)): (x + y).

[1]For the unknown reason to me, proving expression with the \natural type is harder than proving for \integer type limited by a condition x >= 0. Therefore, it is not used unless it is necessary.

2.3.2 Logical operators

In addition to standard logical operators such as AND (&&), OR (||) and negation (!), VCC adds another three that are frequently used in logic. Both operands must be integral:

- $\Longrightarrow$ operator – x $\Longrightarrow$ y iff !x || y
- $\Longleftarrow$ operator – x $\Longleftarrow$ y iff x || !y
- $\Longleftrightarrow$ operator – x $\Longleftrightarrow$ y iff x $\Longrightarrow$ y && y $\Longrightarrow$ x

2.3.3 Quantifications

A simple expression may be extended by quantifiers that has the same purpose as classic quantifiers in a predicate logic. The quantified expression has form \Q T v; E, where \Q is a quantifier, v is a bound variable of type T and E is an expression with the bound variable. The scope of the variables extends to the end of the expression. There are 3 quantifiers:

- \forall represents $\forall$ quantifier, i.e. it returns \true iff the expression E is evaluated to non-zero for every possible value of the type T
- \exists represents $\exists$ quantifier, i.e. it returns \false iff the expression E is evaluated to zero for every possible value of the type T.
- \lambda represents λ quantifier, i.e. it returns a map m from type T to type of expression E such that m[v] == E.

2.4 Types

VCC has a enriched type system at its disposal. Like the C language, it divides types to a value type and object type. Object types are compound types (struct or union types), array object types, \threads, \claims, and \blobs (not used in our work), all other are value types. Claims are described in separated section. We will need to know only one thread value, the current thread that is represented by global variable \me. The "object" term is a bit overloaded in VCC. VCC has a type \object corresponds to C pointer, whereas the word "object" refers to the type. The function \non_primitive_ptr determines whether the type of the pointer parameter is an object type or value type.

2.4.1 Map type

We have already met with map when discussing lambda quantifier. Map type represents standard mathematical function from type T1 to type T2. There is a restriction that both types must be value types. Its syntax is similar to C arrays, but arrays use indices. We can declare variable or object field T1 m[T2], so the type of m is T1[T2]. Then, for every expression E of type T2, the m[E] is an expression of type T1.

2.4.2 Set of pointers

VCC also defines type \objset as set of pointers. It is semantically equal to map definition typedef \bool \objset[\object], but is treated as different type. Set is very useful mainly for ownership mechanism, where an objects owns a set of other objects. Dealing with set is easier because there are defined many set operations with infix operators:

- Detection whether two sets are disjoint (\disjoint), set is subset of another (\subset) and object is member of a subset (\subset).
- Appending object to set (+) and removing object from set (−)
- Creating of intersection (\inter) and difference (\diff) of two sets
- \universe - Set of all objects in the model.

2.4.3 Objects

Function	Description
\objset \span(\object)	Set that consist of object itself and from pointers to all its primitive fields
\objset \extent(\object)	All pointer from \span and all struct fields
\objset \full_extent(\object)	All pointer from \extent
\object \embedding(\object)	Embedding of field such that o = \embedding(&o−>f)

Table 2.1: Functions for fields of objects

Every object in VCC model is defined such that it has set of fields and it may define invariants. If we get a pointer to a field f of a value type T, then pointer in the form &o−>f is a pointer of type T and o is the embedding of the field. Objects have also predefined ghost field. All the fields can be managed together, as we can see in table 2.1.

If we want to argue about objects, we must define something like a "good state" of the object. The object is in a good state if it has only good properties called *object invariants*. VCC allows to define arbitrary number of invariants. However, when the object is created, invariants do not hold, we do not know anything about the it. At the beginning, all object are *open*. If we fulfill all the invariants or if we know that all invariants hold, the object is said to be closed. Every object has field \closed that indicates it. Note that !\closed, invariants may hold, but we do not know it for an unknown object.

Wrap/Unwrap protocol

Every object has an owner. The owner of the object is kept in field \owner. Threads are objects too, while thread always owns itself, so they are always the roots of hierarchy. An object that is owned by the current thread is said to be *wrapped* if it is closed, and *mutable* if it is open. Mutable object are not so interesting because only the thread can manipulate them and we have no

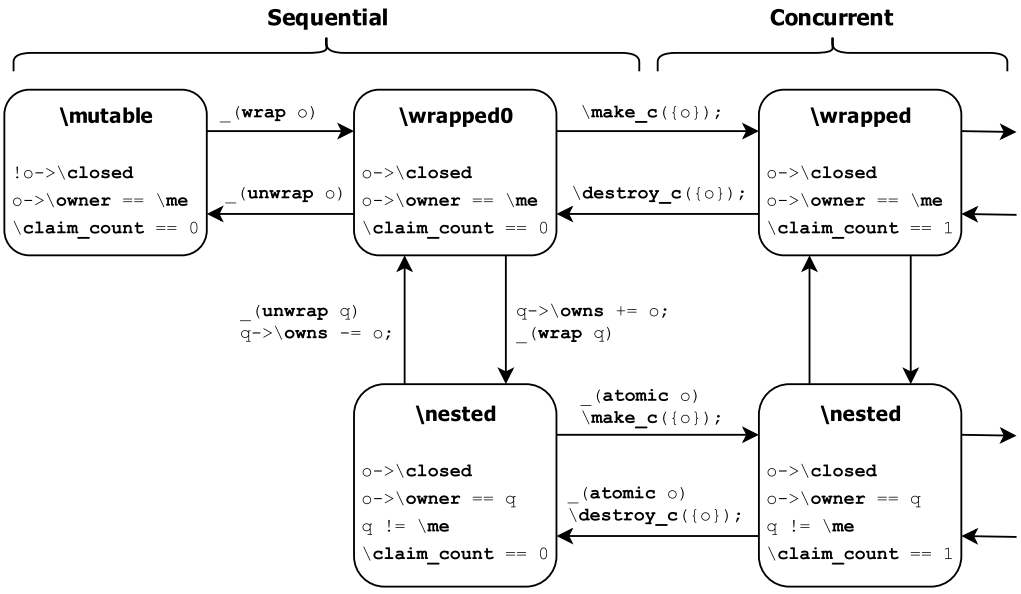


Figure 2.1: Wrap/Unwrap protocol.

guarantee that invariants hold. All new objects and local variables are mutable. Moreover, no other may own the mutable object, only threads may.

We presents the protocol that transforms the mutable object to a wrapped one and back. There are two statements that perform the wrap/unwrap protocol on object o: _(wrap o) and _(unwrap o). First we present wrapping:

1. Assert that object o is mutable
2. Assert that all objects whose ownership is to be transfered to o are wrapped and writable
3. Assign o−>\closed = \true;

And now we present unwrapping:

1. Assert that object o is wrapped
2. Assert that object o is writable
3. Assume that invariants hold (they actually holds, but VCC must assume it explicitly)
4. Assign o−>\closed = \false
5. Add the span of the object (all its fields) to the writes set;
6. Set \me the current thread to be the owner of any objects owned by o

The objects may be owned also by other object. If we want to express that an object o1 wants to own object o2, we must define invariant in o1that contains expression \mine(o2). It says that object o2 is included in o1−>\owns, the set of objects owned by o2. The wrapping protocol is the same as for the threads. The

object owned by another object is then called nested. If the invariant with \mine is more complicated, we may define the type of object o1 with _(dynamic_owns). However, we lose automatic handling with \owns sets. We may see the whole diagram of wrapping protocol on picture 2.1. The table 2.2 show all possible object states.

Object annotation	**Description**
\bool \wrapped(\object)	o->\owner == \me && o->\closed
\bool \wrapped0(\object o)	As wrapped and o->\claim_count == 0
\bool \thread_local(\object)	The object is in the domain of the current thread
\bool \mutable(\object)	o->\owner == \me && o->\closed
\bool \nested(\object)	Owner of the object is not a thread

Table 2.2: Object annotation that express all possible states of object

2.5 Function Contracts

The specification of a function is called contract, because it gives obligations on both the function and its callers. There are classic precondition and postcondition statements and writes clause that enables writing to the pointers. The summary is placed in table 2.3.

Function contracts	**Description**
_(requires :pure b)	Precondition of the function
_(ensures :pure b)	Postcondition of the function
_(writes :pure \object o, ...) _(writes :pure \objset s)	It gives the right to write to a set of pointers
_(maintains \bool p)	Precondition and postcondition together
_(updates \object o)	Precondition and postcondition with write permission

Table 2.3: Function Contracts

2.6 Global and static data

Global and static (visible only inside its module) variables are part of program memory that is usually called data segment. These variables are created and initialized at start of program. Global variables have the feature that are shared between function calls. They are allocated for the continual program execution and may be accessed by more threads. In functional analysis, every function may read from and write to them. These variables are often set as volatile and updated in atomic blocks. Otherwise, we must explicitly guarantee that no more than one thread access them in one moment. Claims are often used for this purpose.

Global and static variables are initially considered mutable at the entry point of the program because there is convincingly only one thread and it must be able

to own every part of accessible memory. The entry point of C programs is the main function and VCC annotates this function with the contract program_entry_point(). This function implicitly takes a state as a hidden parameter and set it as the current entry point state.

Because of the nature of primitive objects (basic data types), the primitive variable cannot exist alone in verifier model. Therefore for every global primitive variable v, there is a dummy anonymous object with only one field v. The object may be reached by the expression \\embedding(v). For more complicated and multi-threaded programs, it is more suitable to use ownership mechanism where there exists a ghost object representing the global state and owning all the global variables or its embeddings, or better, user objects own them.

In the case of arrays, there is one owner for all elements if the array consists of primitive types, the embedding is the same for all the elements. This is opposite to an array of structures, where every element of array is a separate object. Unfortunately, there is an inconsistency if the array of non-primitive objects is a ghost. Then, it behaves strange and there is no change to reason about every element separately because individual elements may be mutable, but cannot be wrapped. Actually, trying to annotate function with the precondition that element is even thread local, makes the specification inconsistent. We must introduce a global structure and encapsulate array the inside. Then, it behaves correct[2], but to allow the thread to write one element, it must also own the structure, and it is not good for concurrency proving.

2.7 Claims

Claims are mechanism how to promise that object's invariants hold and the object is closed. If we have a shared resource, we may access it only when we are sure that no other threads modifies it. The claim is an object that has a special invariant. It states that a set of objects is closed. Since it is the invariant, the property applies only when the claim itself is closed. The question arises how we know that claim is also close. It may either be wrapped or claimed by another claim. Since claiming mechanism is not needed always, only types annotated with _(claimable) may have claims on themselves. In the table 2.4, there are some operations and expression with claims.

2.8 Termination

A function, block, or loop may be annotated with a _(decreases) annotation. This annotation guaranties that the block of code terminates. The annotation have variable parameters that are all \natural. The sequence of expressions is a function from states of the body or block to finite tuples of \naturals. If the function is monotone and decreasing, the VCC is able to prove that it terminates. The simple example is a for loop, where iterator increasing. Then the expression in decrease annotation may be just the iterator. The simple functions that have no recursion may define _(decreases 0) that is absolutely sufficient.

[2]This is probably a bug because there is no reason for this behavior, especially when it has right semantic inside a ghost structure.

Claim annotation	Description
\bool \claims_object(\claim c, \object o)	It states that claim claims an object
\bool \account_claim(\claim c, \object o)	It is alias to \wrapped(c)&& \claims_object(c,o)
\claim \make_claim(\objset s, \bool p)	This command creates a claim, The claim claims all object in set s. The objects must be wrapped, so all invariants hold. Because creating a claim on an object o assigns to o−>\claim_count, it requires write access to the object o.
\destroy_claim(\claim c, \objset s)	It is opposite operation to make_claim
\bool \active_claim(\claim c)	It activates the claim
_(by_claim \claim c)	Useful for reading of non-volatile fields
o−>\claim_count	Number of claims on object
_(always \claim c, \bool cond)	

Table 2.4: Claim annotation

3. FAT file system

File Allocation Table (*FAT*) [21] is a file system or family of similar file systems that has an origin in the late 1970s. The first version dates back to the year 1977 and it was still 8-bit FAT, originally intended to be a file system for Microsoft Disk BASIC interpreter. The file system was the first introduced in Microsoft's DOS operating system in 1980 as the new version of 12-bit FAT, called FAT12. It was then used in all versions of PC-DOS and later MS-DOS, and also all versions of Windows that was based on DOS. Windows systems may still read it, but starting with Windows NT, it has been replaced by its successor, NTFS file system.

The file system was created for the widespread personal medium of the era, floppy disks that could be formatted under DOS and later Windows and some other systems. Their size is measured in kilobytes and soon started to be insufficient for their small capacity. It has come the time of hard disks, but the first version was not able to address large mediums. The file system had to be extended to new versions and soon it was adapted very slightly to larger mediums. Now there are three commonly used major variants: FAT12, FAT16 and FAT32. Their architecture is very similar, but they differ in maximum number of allocation units, called clusters.

Today, the file system is used in a wide range of mediums. It is supported in almost all operating systems for PC and for compatibility reasons, it is well-suited system for data exchange between devices. We may still find it on floppy disks and hard disks as well as in many embedded and portable devices. It includes devices with flash-based memory like USB flash drives, solid-state drives and memory cards for mobile phones, digital cameras and other electronics.

For floppy disks, FAT system of types FAT12 and FAT16 with short 8.3 filenames has been standardized in ECMA-107[11] as system for data exchange. Even disks without FAT installed may have standard PC disk partition scheme, which uses a master boot record (MBR), originally introduced in MS-DOS 2.0 with FAT12 as the first sector and included in Extensible Firmware Interface (EFI) specification.

The architecture of the system is very simple. It consists of a set of singly-linked list of clusters in a table at the beginning of the partition. It means that the firmware for it has usually small footprint, uses little memory and is easy to implement. It also has not bad performance, but it cannot be compared to newer systems based on faster structures like b-trees. However, simplicity is also its disadvantage. It often accesses storage instead of cached memory, the FAT table is fixed on partition start and it is accessed very often, loading of large files may be ineffective because of slow operations with linked list and fragmentation, small files cause slack space, etc. Because of that it does not provide the same scalability and reliability as modern systems.

DOS is no longer widely used operating system, except simpler devices, and driver in Windows is naturally proprietary. In order to allow other systems to access the FAT file system, there has been implemented more FAT file system drivers. We can mention, for instance, vfat driver, which is de facto standard in access to FAT on Linux. The typical scenario is such that the new file system

driver is associated with some operating system, as it is the case of FreeDOS or ReactOS.

There are also open-source file systems, independent of operating system like Renesas Electronics' M3S-TFAT-Tiny[25], Freescale MQX™ File System[19] in Freescale MQX™ RTOS, FullFAT[13] or libfat[6] for Nintendo. For verification, we have chosen ChaN's *FatFs*, a generic FAT file system module dedicated to for embedded systems. In the following sections, we briefly describe the basics of FAT file system and the architecture and implementation of FatFs.

3.1 FAT file system architecture

The file system assumes to have the facility to block access, which was characteristic for the medium at the time of its creation. The smallest block which can be accessed is called sector. Sectors are numbered with Logical Block Addressing (LBA) that is a simple linear addressing scheme, unlike abandoned Cylinder-Head-Sector (CHS), which was used for hard drives. The size of these sectors may vary and depends mainly on the abstraction provided by the device. The smallest sector size is 512 bytes.

As most file systems, it does not utilize individual sectors because it is not usually effective to split file into such small blocks. Moreover, how the capacity of mediums grows, it is more and more difficult to keep and seek so many small pieces of data. As the name suggests, the FAT file system utilize a table with indices pointing to allocation units of the same size that are called clusters. Every cluster consists of a number of sectors, but this number is not fixed and every concrete file system may set it by its own, depending on size of medium. The larger volume is, the larger cluster is appropriate to choose. Then every file in the system is list of clusters that are stored in the FAT.

Region	**Description**	**Size (in sectors)**
Reserved Region	Volume Boot Record (VBR)	Defined in boot sector (one in FAT12/FAT16)
	FSINFO sector	
	Vendor defined sectors (e.g. copy of boot sector)	
FAT Region	Original File Allocation Table	(number of FATs) * (sectors per FAT)
	Optional number of FAT copies ...	
Root Directory Region	Array of 32-byte directory entries	(number of entries * 32) / (bytes per sector)
Data Region	Array of data clusters	(number of clusters) * (sectors per cluster)

Table 3.1: FAT partition is divided into four regions.

Every FAT file system partition contains four distinct regions described in the table 3.1: Reserved region, FAT region, root directory region (only in FAT12, FAT16) and file and directory data region. We briefly describe these regions with

emphasis on the parts that we want to analyze. There are well-defined structure with all important information. We must realize that all data are stored little-endian. Some platforms, like Intel x86 processors, may read it directly, but other must reverse the byte order.

3.1.1 Reserved region and Boot record

The term boot sector is a little confusing because there are two types of the sectors: Master boot sector (MBS) and Volume boot sector (VBS). The very first sector (with index 0) is MBS. When the computer is turned on, it is the first sector that is read. Moreover, if it contains bootstrap code, it is responsible for loading of the operating system. The sector is not part of any file system, but its purpose is to detect and select the proper file system in partition table that carries.

The table contains four entries. Not every entry can be occupied, but when it is, it should contain a valid number of sectors which is the first one of the corresponding partition. Note that some mediums have no MBR, such as floppy disks or optical discs, because there is always only one partition. The file system driver must be able to detect it.

Name	**Offset (byte)**	**Size**	**Description**
BPB_BytsPerSec	11	2B	Sector size in bytes. Possible values are 512, 1024, 2048 or 4096.
BPB_SecPerClus	13	1B	Sectors per cluster It must be greater than 0 and power of 2.
BPB_RsvdSecCnt	14	2B	Number of reserved sectors
BPB_NumFATs	16	1B	Number of FAT copies. It must be at least 1. There are usually 2.
BPB_RootEntCnt	17	2B	Number of root directory entries for FAT12/16 (0 in FAT32)
BPB_TotSec16	19	2B	Sectors in volume (FAT12/FAT16 only)
BPB_FATSz16	22	2B	FAT size in sectors (FAT12/FAT16 only)
BPB_TotSec32	32	4B	Sectors in volume (FAT32 only)
BPB_FATSz32	36	4B	FAT size in sectors (FAT32 only)
BPB_RootClus	44	4B	Root directory first cluster
BPB_FSInfo	48	2B	Offset of FSINFO sector (only FATFS)
BS_FilSysType BS_FilSysType32	54, 82	8B	File system type may indicates that there is FAT file system when it starts with "FAT" string.
BS_55AA	510	2B	Signature word (same for MBS and VBS)

Table 3.2: Boot Sector and BPB Structure

Every partition contains its own VBS. VBS is located at the beginning of a par-

tition in the first sector of reserved region. It contains a bootstrap program (not useful for our purpose) and all important data about file system that we need. The basic parameters are stored in BIOS Parameter Block (BPB), other after it. BPB was introduced because it contains all the relevant information for the boot loader. Since BPB always starts at the same offset, all its parameters have known location. The data organization is a bit confusing because it may differ depending on version of the data block and file system type. However, according to specifications, there should always be defined new BPB fields for either the FAT12/FAT16 or FAT32 BPB type. In the table 3.2, there are all parameters that we use.

Since file system is only a table with linked lists, there is no natural mechanism how quickly determine the free space on volume and any free cluster. It is a problem mainly for FAT32 because its FAT may be large. Therefore, FAT32 has introduced FS information sector with structure shown in table 3.3. It is usually placed in the sector right after boot sector. It contains information about the number of free clusters and last allocated cluster. However, this information is not reliable and should be taken only as a hint.

Name	**Offset (byte)**	**Size**	**Description**
FSI_LeadSig	0	4B	Signature 0x41615252 validates FSINFO.
FSI_StrucSig	484	4B	Signature 0x61417272 validates FSINFO.
FSI_Free_Count	488	4B	The last known free cluster count or value 0xFFFFFFFF when it is not known. It should be range checked to make sure it is valid.
FSI_Nxt_Free	492	4B	The cluster number at which we should look for free cluster or 0xFFFFFFFF if it is invalid. By default, FAT driver starts to looking at the first cluster.
FSI_TrailSig	508	4B	Signature 0xAA550000 validates FSINFO.

Table 3.3: FS Information Sector introduced in FAT32

3.1.2 FAT and directory structure

All user data are stored in files. The FAT file system introduces the concept of clusters. Every file is stored in a chain of clusters, so a file consists of at least one cluster and no two file may share one cluster. The cluster chain may not be continuous on the disk, so we connect it with a linked list. Such a linked list is stored in the FAT. Every cluster has one integer entry in the table. The number in the entry can signify principally three things:

- Number of the next cluster of a file. This value is also index into FAT and the entry on the index is dedicated just to the next cluster.

- The last cluster in file (end of cluster chain - EOC). It indicates the current cluster is the last cluster of the chain. The value in the entry is recognized as EOC, because it is larger than any possible cluster number for the given FAT sub-type.
- Free cluster. If the entry is zero, there are no data in the cluster.

Now we have files, but in order to access a file, we need to know the first cluster of chain. For this reason, there are directories. A directory is a special file that contains directory entries. Every directory entry occupies 32 bytes and keeps principal information about a cluster chain. The structure of the entry is given in the table 3.4.

Name	**Offset (byte)**	**Size**	**Description**
DIR_Name	0	11B	8.3 short filename. The name cannot contain some special characters and it must not be empty
DIR_Attr	11	1B	File attributes, e.g. ATTR_READ_ONLY, ATTR_VOLUME_ID and ATTR_DIRECTORY
DIR_FstClusHI	20	2B	Higher 16-bit of this entry's first cluster number (always 0 for FAT12/FAT16).
DIR_FstClusLO	26	2B	Lower 16-bit of this entry's first cluster number.
DIR_FileSize	28	4B	The size of file in the cluster chain (cached value)

Table 3.4: FAT directory

We must be able to distinguish between files, so every entry has a name dedicated to the file. Directory entry may keep the first cluster to cluster chain with another directory. This means that there can exist a directory tree and every file has its own path that starts in the root directory. FAT12 and FAT16 have fixed size root directory as one of FAT regions, in contrast with FAT32 where the root directory is an ordinary directory from data sector. Anyway, both of them have the same format.

Note that directory is just an array of entries and every entry has its index. If we have the first sector and index of the entry, we are able to calculate number of clusters, absolute number of sectors, position of sector relative to the cluster and the exact address where the entry is in the sector.

3.2 FatFs module

FatFs[5] is a FAT file system driver designed for small embedded systems. It is an open source project under BSD-compatible license and so it may be incorporated into proprietary products. It is generic and it can be independently ported to various types of hardware because its implementation is fully separated

from any underlying device. There is also a subset of the driver called Petit FatFs for tiny 8-bit microcontrollers.

It is a good choice for small microcontrollers with limited resource, such as 8051, PIC, AVR, ARM, Z80, 78K, etc. It is capable of handling diverse small mediums as SD cards. Microsemi Corporation has included it into its SmartFusion® system-on-chip (SoC) device[20]. It also became a common choice for amateur engineers and hobbyist.

There are some features and advantages over other file system drivers:

- Open-source software with license equivalent to BSD 1-Clause License, that is open for education, research and commercial development.
- Compatible with original FAT file system formatted on Windows and DOS-based operating systems.
- Actively maintained.
- Very portable, written in ANSI C (C89) that is supported by almost all C compilers.
- Independent of storage device and operating system. It provides public disk I/O layer interface and its implementation must be written for every system separately.
- Independent of third-party software component.
- Very small footprint of compiled binary and very little memory usage.
- Highly configurable, a lot of functionality is optional and may be changed or disabled.
- Various features including:
  - Supported all three common types: FAT12, FAT16 and FAT32.
  - Possible to mount up to 10 volumes (physical drives and partitions too).
  - Multiple ANSI/OEM code pages including double-byte character set (DBCS).
  - Long file name support in ANSI/OEM, UTF8, UTF16 or Unicode.
  - Ability to format a new FAT partition and function FDISK.
  - Support for FS information sector.
  - RTOS support for multi-task operation, locks and thread safe functions.
  - Read-only, I/O buffering, etc.
- There are no unnecessary limitations:
  - Unlimited number of open files (limited by memory).
  - All supported sector sizes (512, 1024, 2048 and 4096 bytes).

- Cluster size up to 64 KB with 512 bytes/sector.
- No limit for number of clusters in FAT12 (2^{12} clusters), FAT16 (2^{16} clusters) or FAT32 (2^{28} clusters) type.
- File size up to 4 GB (depends only on FAT configuration).
- Volume size up to 2 TB with 512 bytes/sector.

The module is developing under Windows platform and it can be compiled with Microsoft Visual Studio. With all the mentioned advantages, it is very eligible for verification in VCC that is running on Windows architectures and has shared parts with Visual Studio compiler. Since the source code has no dependencies on a concrete device or additional library, it eliminates problem with defining of specification for a complicated interface. Many features may be also disabled. It allows us to concentrate on the analysis of bearable small part of the system.

3.2.1 Architecture

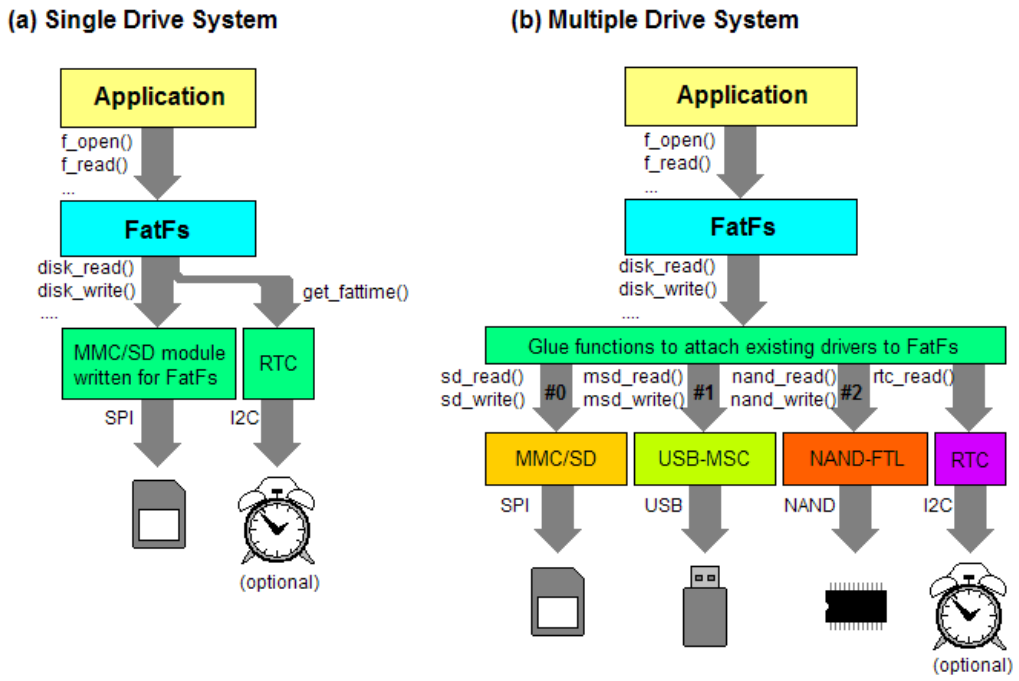


Figure 3.1: FatFs architecture.

The architecture of file system is very simple. There are two interfaces provided by the module: Application interface for user applications and Device Control interface needed by the file system to access a device. The architecture is demonstrated in the diagram 3.2. The file system also defines three main data structures: FATFS, FIL and DIR. Both interface and structure are well documented that facilitates the system usage and also creation of specification.

Application interface

The interface defines a set of basic functions to control file system and describes what operations it can perform with FAT volume. It is very similar to C Standard

Input and Output Library and therefore it may be used directly in an application or easily integrated to an OS. We are interested in only a few of these functions. Here is the list:

- f_mount - Register/Unregister a work area
- f_open - Open/Create a file
- f_close - Close an open file
- f_read - Read file
- f_write - Write file
- f_lseek, f_truncate, f_rename - Edit files
- f_opendir, f_closedir, f_readdir, f_mkdir - Operations on directories
- f_stat, f_chmod, f_utime, f_tell, f_eof, f_size, f_error - File and directory statistics, error detection
- f_chdir, f_chdrive, f_getcwd - Relative path feature
- f_gets, f_putc, f_puts, f_printf - Strings and characters functions
- f_getlabel, f_setlabel - Get or set volume label
- f_sync, f_forward, f_unlink, f_getfree, f_mkfs, f_fdisk

Device Control interface

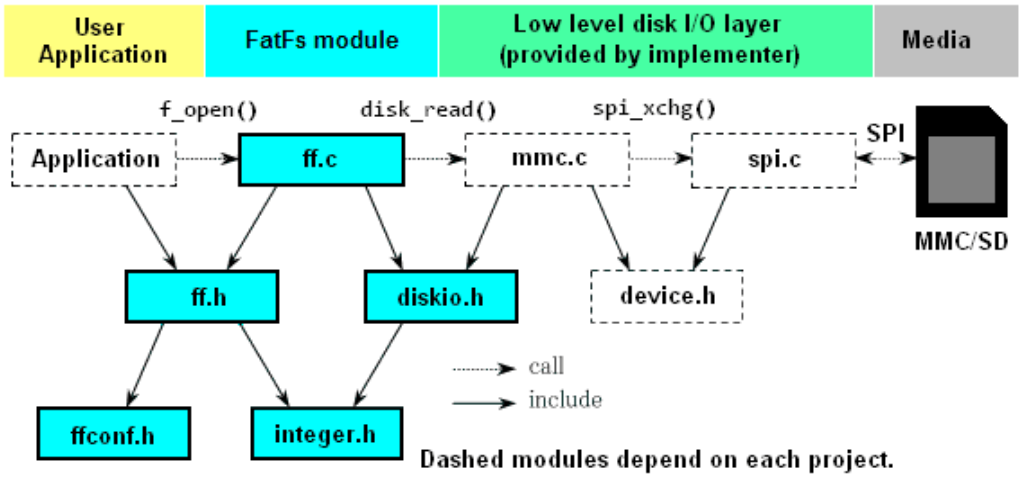


Figure 3.2: The dependency diagram shows the typical configuration of an embedded system which includes FatFs module.

The disk interface consists of only six functions that are designed to communicate with an underlying device. FatFs access the device through this API that is very similar to POSIX interface for device I/O control. As we can see in the diagram 3.2, disk driver implementation is not part of the module. A typical

scenario is that the device already has a driver and the API implementation is just a wrapper over it that only connects it with FatFs. There are some examples of driver for various implementations on FatFs webpage. Let's mention that the get_fattime is usually not part of the disk driver, but it is supported by OS. Here is the list of functions:

- disk_status – Get device status
- disk_initialize – Initialize device
- disk_read – Read sector(s)
- disk_write – Write sector(s)
- disk_ioctl – Control device dependent features
- get_fattime – Get current time

FATFS structure

The FATFS structure (file system object) represents a dynamic work area of single logical drive (volume). The application provides the structure and it is (un)registered into FatFs with f_mount function. The module does not care about allocation mechanism. The initialization is realized immediately in f_mount, during the first call of API function that access the volume (delayed initialization), or when media changes.

Even structures for file and directory have a link to FATFS to be able to access the information. Therefore, the only function f_mount gets a FATFS object as a parameter, the other functions get it from FIL or DIR structures or from a static storage of registered FATFS. It is assumed that the application code does not modify the structure fields directly (the principle of encapsulation). This also applies for the other structures.

```
typedef struct {
  BYTE  fs_type;      /* FAT sub-type (0:Not mounted) */
  BYTE  drv;          /* Physical drive number */
  BYTE  csize;        /* Sectors per cluster (1,2,4,...,128) */
  BYTE  n_fats;       /* Number of FAT copies (1,2) */
  BYTE  wflag;        /* win[] flag (b0:win[] is dirty) */
  BYTE  fsi_flag;     /* FSINFO flags (b7:Disabled, b0:Dirty) */
  WORD  id;           /* File system mount ID */
  WORD  n_rootdir;    /* Number of root dir. entries (FAT12/16) */
#if _MAX_SS != _MIN_SS
  WORD  ssize;        /* Sector size (512,1024,2048 or 4096) */
#endif
#if !_FS_READONLY
  DWORD last_clust;   /* FSINFO: Last allocated cluster */
  DWORD free_clust;   /* FSINFO: Number of free clusters */
#endif
  DWORD n_fatent;     /* Number of FAT entries (#clusters + 2) */
  DWORD fsize;        /* Sectors per FAT */
  DWORD volbase;      /* Volume start sector */
  DWORD fatbase;      /* FAT area start sector */
  DWORD dirbase;      /* Root directory area start sector */
                      /* (FAT32: Cluster index) */
```

```
  DWORD database;      /* Data area start sector */
  DWORD winsect;       /* Current sector appearing in the win[] */
  BYTE  win[_MAX_SS]; /* Disk access window for directory, */
                       /* FAT (and file data a tiny cfg.) */
} FATFS;
```

Source code 3.1: File system object structure (FATFS)

Fields of the structure 3.1 are usually self-explaining. Fields csize, n_fats, n_rootdir, ssize, last_clust, free_clust, fsize and volbase correspond exactly to the values stored in Boot sector. Number of clusters defines exactly the FAT type, as it is described in FAT specification. Field fs_type identifies the sub-type of FAT (positive number) or it is zero, which means that FATFS object and device are not initialized yet. Fields volbase, fatbase, dirbase, database are the first sectors of every file system region (where dirbase is the first cluster of the root directory in FAT32). Other fields are specific for the implementation.

Every registered file system object is bound to a logical drive number. The number is stored in drv. By default, each logical drive is bound to the physical drive with the same drive number. Then, the first valid FAT partition is mounted. In multi-partition configuration, there must be defined volume management table that relates the logical drive number to physical drive and partition. Every newly registered object gains an id number that serves as identification with files and directories object and simple versioning. Field fsi_flag is a bit-vector that indicates whether the module trusts FSINFO structure and whether last_clust, free_clust fields are synchronized.

Since the module may work with only one disk segment at the same time, we can talk about the current sector. It uses a cache of fixed size called disk access window that is part of FATFS structure and winsect field keeps a number of the current sector. The sector is loaded into cache by disk_read call. If the module needs to read another sector, it must "move the window" by calling disk_read with another sector number as parameter. The windows may be also saved by calling disk_write. The usual operation is sector modification - reading a sector, data editing in the window and writing the sector. To avoid unnecessary disk writing, wflag field indicates whether the window is dirty.

FIL structure

Both FIL and DIR structures share the first two fields. It permits us to perform a generic check of validity without knowing the exact type of the object. Field id is useful when we unmount and mount the same FATFS object because all previously open files are invalidated. Fields sclust and fsize are copied from directory entry.

```
typedef struct {
  FATFS* fs;      /* Pointer to the related file system object */
  WORD   id;      /* Owner file system mount ID */
  BYTE   flag;    /* Status flags */
  BYTE   err;     /* Abort flag (error code) */
  DWORD  fptr;    /* File read/write pointer (Zeroed on file open) */
  DWORD  fsize;   /* File size */
  DWORD  sclust;  /* File start cluster (0:no cluster chain,
                  /* always 0 when fsize is 0) */
  DWORD  clust;   /* Curr. cluster of fpter (not valid if fprt=0) */
```

```
  DWORD  dsect;  /* Sector number appearing in buf[] (0:invalid) */
#if !_FS_TINY
  BYTE   buf[_MAX_SS]; /* File private data read/write window */
#endif
} FIL;
```

Source code 3.2: File object structure (FIL)

DIR structure

Directory is a special type of file, so it also has field sclust with start cluster. The structure is used by algorithms that look for a file or directory name in directory entries. The current state of the algorithm is determined by index of the current entry. All the fields clust, sect and dir are calculated from the index and the start cluster number. Field fn is a pointer to an array allocated by FatFs (the allocation mechanism depends on configuration) that contains searched file name. It is compared to name stored in the directory entry accessible using the dir pointer to the fs->win, the currently cached sector.

```
typedef struct {
  FATFS* fs;      /* Pointer to the owner file system object */
  WORD   id;      /* Owner file system mount ID */
  WORD   index;   /* Current read/write index number */
  DWORD  sclust;  /* Table start cluster (0:Root dir) */
  DWORD  clust;   /* Current cluster */
  DWORD  sect;    /* Current sector */
  BYTE*  dir;     /* Pointer to the current SFN entry in the win[] */
  BYTE*  fn;      /* Pointer to the SFN (in/out) */
                  /* {file[8],ext[3],status[1]} */
} DIR;
```

Source code 3.3: Directory object structure (DIR)

3.2.2 Implementation

There are some implementation details in FatFs that are good to mention for better understanding how the module works internally.

Data types

Data structures of FAT file system have fixed size of all its entries in bytes. These integer types have name BYTE for 8-bit **char**, WORD for 16-bit **short int** and DWORD for 32-bit **long int** in WinAPI. FatFs module assume these sizes. It also assumes basic integer size 16 or 32 bits (VCC has a 32-bit **int**). All of these definitions are set `integer.h`. If the platform does not support the exact sizes, it must be resolved with care.

Configuration options

FatFs is highly configurable. Configuration is placed in `ffconf.h` file as a set of preprocessor macros and there is a lot of conditional compilation in the source code. About 20 distinct constants can be edited, we show an example of only a few of them:

- _VOLUMES – Maximum number of volumes available in FatFs.
- _MIN_SS/_MAX_SS – The range of possible sector size. If they are not the same, the file system must ask a device, what size of sector supports.
- _FS_READONLY – Setting forbids to edit the file system of a device.
- _CODE_PAGE – It specifies ANSI and OEM code page.
- _FS_RPATH – Relative path feature.
- _FS_NOFSINFO – It enables use of FSINFO structure.
- _MULTI_PARTITION – If set each logical number represent arbitrary drive/-partition, otherwise number is combined with corresponding drive and one FAT partition is supported.
- _FS_LOCK – File lock control feature, it indicates the number of files/sub-directories that can be opened simultaneously.

File path

A common way to represent a location in file systems is to organize files in the directory tree hierarchy. Then, every file may be found in a path from the root directory over all directories. The path is usually a string that consists of all the directories separated by delimiter, which is usually `'/'` character. The FAT file system has the well-known format of the path: `[drive letter:]\directory\file`. The drive in the path is an uppercase letter in the English alphabet, so DOS systems can have only 26 distinct partitions. Delimiter is `'\\'` that differs from UNIX systems (because `'/'` is used for command line options).

FatFs has slightly different regexp path pattern:

```
(\[drive#:\])?(/|\\)*directory(/|\\)+file
```

Instead of letters, number is used, so only 10 disks or partitions can be mounted. If the drive number is omitted, it is assumed as default drive (drive 0 or current drive). There are some other limitation:

- Separator can be either `'\'` or `'\\'`, root separator can be omitted and duplicated separators are skipped.
- Control characters from `'\0'` to `'\0x1F'` are recognized as the end of the path name.
- Space is recognized as the end of the path name.
- Trailing spaces and dots at the end of path are ignored.
- If relative paths are disabled, dot directory names are not allowed, leading separator is ignored and can be omitted. The default drive is 0.

4. Verification of implementation

This chapter describes the verification of FatFs. We have chosen the version FatFs R0.10b, the most current version at the time of verification. The module enables many configurations. We have enabled configurations _VOLUMES=10, _CODE_PAGE=1, _MIN_SS=512, _MIN_SS=4096, _FS_NOFSINFO=0, other are disabled. There are verified all functions for the configuration except f_read. Note that we do not verify the FAT file system because we do not represent medium as reliable, but the code of FatFs module.

There is a list of verified functions: f_open, f_close, f_mount, mem_cpy, mem_set, mem_cmp,chk_chr, move_window, clust2sect , get_fat, dir_sdi, dir_next, ld_clust, create_name, follow_path, get_ldnumber, check_fs, find_volume, validate. Several of them are also verified for _FS_READONLY=0 and there are also verified some other functions with this configuration: sync_window, sync_fs, put_fat12, remove_chain, st_clust.

Several of these functions has been divided into smaller function because they were too complicated for VCC. The functions dir_find, follow_path_wrapped and initialize_fs_type_and_limits are unstable. Their function contract is correct and they are verified. However, their verification process may not terminate. They are commented out with _VCC_UNSTABLE macro.

4.1 Source code modifications

Although one of the goals of the verification is to analyze the source code as it, unfortunately, it was not possible to avoid both marginal and significant modifications. As a modification, we can also consider bug fixes, but these are the desired changes, of course. Some changes were possible (or necessary) to perform before the verification, some of them has proved to be necessary during the process.

Some of these changes may be found by smarter compiler or static analyzer. Generally, it is better to use less sophisticated method to detect bugs before verification because every bug found with verification is valuable, but its detection takes more time. The biggest modification was splitting of multiple functions into smaller functions, which is always described in the particular case. Besides these changes and errors found, we show other modifications of the code. Most of the changes are marked with a specific comment that begins with the appropriate keyword:

- _MODIFIED – Modifications that are not necessary but improve the code or facilitate verification.
- _FIX – Fixes of bugs that are found during the verification.
- _BUG – Found bugs that have not been fixed.
- _VCC – Changes forced by VCC. They should not change semantics of the program.

4.1.1 Conditional compilation

There were added conditional compilations using configuration macros in appropriate places. Conditional compilation can help verifier to deduce information better. Regular code must be included into the model, but preprocessor can discard pieces of code before the compilation.

The constant values are often testing inside conditions. However, constant expression may be deduced during compilation, so it is better to conditionally compile this expression. Conditional compilation is sometimes useful to suppress compiler warnings. When some code is erased by preprocessor, the compiler can afford better static analysis and it can detect more warnings, e.g. unused or uninitialized variables. VCC may also detect dead code and the conditional compilation is the only solution. There has been found even whole **if** branches that could be eliminated by a constant condition, i.e. constant such as _DF1S (constant indicating multi-byte character) or _MULTI_PARTITION. Also fields wflag and fsi_flag of FATFS are not needed in read-only configuration.

4.1.2 Variables scope reduction

Scope reducing follows the principle of locality claiming that variable declaration should come as close as possible to its use in a subroutine. ANSI C does not allow to declare variable elsewhere than at the beginning of the block, even so it can be reduced into more nested block. We reduce variables declared in a scope that is larger than it is necessary.

These changes are not needed for proper verification, but they help when writing an annotation. the difficulty of verification increases roughly exponentially with the code length. Reducing some variables will help us focus on the important code. For instance, if we use a variables in the loop, there is a difference whether it is defined inside or outside the loop body because in the first case, we do not even think whether to mention the variable in loop annotation. If a lot of variables are declared in nested block, it may indicate need for refactoring. Then, the smaller function or module verifies easily. Naturally, the process works both ways, the verification gives hint if variables can be reduced.

There are other reasons for reducing scope of variables, not directly related with our analysis:

- It reduces the state space and eases subroutine validation (useful for model checking, less for verification).
- Variable declaration may take some computational time.
- It clutters the lexical name space, (e.g. iterators i, j, k, etc.).
- New variable introduced in smaller scope can accidentally hide another one in larger scope with the same name.
- Variable may be defined (declared and initialized) in one expression that prevents use of indeterminate variable value.

4.1.3 Const-correctness

Const-correctness is a kind of program correctness that helps recognize mutable and immutable variables. In C language, this is a specification of variables in a form of 'const' keyword that helps compiler to identify type of access to memory in compile time, so it is just a form of type-checking. It makes easier to understand the nature of the data, as data type does. It may also help the compiler to reason about the code and even optimize the resulting program. These changes are not necessary for verification with VCC, but have the same effect as reducing of variable scope.

4.1.4 Repair of sequence conflicts

In C language, there are constructs that do not introduce a sequence point. This means that there is not guaranteed an order of expression evaluation. These places without well-defined sequence points are called sequence conflicts. This behavior may occur when passing multiple expressions as routine parameters or operands of an operation (e.g. plus or minus). For instance, function call fn(inc(&i), inc(&i)) is ambiguous because we do not know which increment will be made first. Thus, one expression may change a variable that is used in other expression. The problem is caused by functions that take a pointer to a variable. Some compilers (including VCC) can detect this ambiguity.

4.1.5 Type conversions

VCC is very sensitive to types of operands in arithmetic. Arithmetic operations do not accept all combinations of operand types, especially operations with one signed and one unsigned integer. If an operation is not well defined by C language, VCC reports an error. It is usually the case of a sub-expression that operates on integer types smaller than **int**. Then, the result is signed and since file system utilize primarily unsigned integers, it causes an error. We must explicitly cast one or both operands which causes change of result's type (but not its value). If we already get signed expression, we must cast it into unsigned integer. This operation is precisely defined in C specification, but VCC is unable to do it. TO_UNSIGNED macro perform correct cast explicitly.

An other problem is variable initialization with an expression that has larger size. Common behavior finds the smallest non-negative value that is congruent to the integral expression, modulo variable range, and the bit representation of the new value is the same as bit representation of truncated expression. However, C specification does not define it precisely and so it does not work during verification. We must truncate the expression with appropriate size explicitly. The code contains LS_BYTE and LS_WORD macros that has been changed to behave correctly, i.e safely cast values to BYTE resp. WORD. Note that VCC behaves differently than compilers usually do. Usual compiler behavior is to report a warning when it trying to store a larger type in a variable a of smaller type. However, VCC must prove that expression does not fit into the variable. If it is so, verification fails, otherwise casting is not needed.

- Number constants have **int** or **unsigned int** type. If we want other type, we must cast them.

- Both sub-expressions of conditional expression must be the same type. VCC enforces this rule, even if both types are accepted by outer operation.
- VCC does not accept cast from fixed-size arrays into **const** pointers. If we want to work with pointer, we must take an address of the first array element. (e.g. &array[0]).

4.1.6 Moved constants and definitions to header files

VCC allows to annotate type declarations with invariants and function prototypes with preconditions and postconditions. Macro constants and functions are used in the annotation and therefore must be already visible in header. However, some of these definitions are necessary only in implementation and they are naturally declared there. We must move all necessary constants and function-like macros before type declarations and function prototypes (even in header itself). There are some of these changes:

- Macros of flags and offset address (e.g. file access control and file status flags, FAT sub types, file attribute bits for directory entry) and multi-byte word access macros are shifted up in `ff.h` header file.
- File function return code enumerate type (FRESULT) is shifted up above function prototypes
- Some macros, such as size of a directory entry (SZ_DIR), are moved from `ff.c`
- All function prototypes of disk API are shifted down at the end of `diskio.h` header file.
- Since we need reasoning about device, some macros must be copied from `diskio.h` into `ff.h` header file. The conditional compilation defines the constants only if they do not already exist, because we do not want redundant definitions in two header files.

4.1.7 Modifications according to VCC needs

Every annotated code must include `vcc.h` header file that defines macros used by VCC. However, the header may not occur in the system. So we say that it is included only if _VCC_VERIFY) constant is defined. `vcc.h` file itself defines macro constant VERIFY that is then used the entire project.

Occasionally, VCC has problem to verify or even compile some languages constructs. Sometimes is necessary to conditionally compile some part of code depending on VERIFY constant. For instance, VCC is unable to verify Windows headers. These headers define standard types that are used in file system code. Instead, provided typedefs for embedded platforms may be used. Other modifications are marked with special comment.

4.2 Architecture

4.2.1 Device Control interface

The FatFs defines a disk input-output interface as a set of public functions, constants that give true meaning to their parameters, and return value types. It is not difficult to write a contract for all the API functions because the interface is clearly documented. Our main task is to ensure that the interface stays persistent, for instance, disk_status function must always return the same value, unless it is changed by disk_initialize call. The only solution is to have a direct access to device data.

We know that the API is not stateless, but it hides information about all disk volumes behind. The state cannot be expressed in C language, since it is protected against users of the interface, but data must be accessible because of VCC reasoning. We show that it is possible to declare a ghost structures that represents the internal state of the interface. The state should express the status of a disk, e.g. whether it is initialized or if it is or is not read-only, the size of volume sector, and other information about the medium that depend on its type we get by calling of disk_ioctl function. There is no generic disk I/O implementation, but we provide a simple dummy implementation, that reflects the function contracts. It allows us to verify the API and show that our contract is valid.

State of interface and private static data

Consider an example: We have a module that consists of a header file with interface and an implementation file 4.1. This is common situation in C language. The data privacy is achieved by declaring a static variables in the implementation part. Only the module itself can access the data because it is not mentioned in the interface. If we want to expose it, we must define a getter that returns the value of the variable. In the case of modifying, a setter is good enough. Now the problem is coming. The getter must always return the same value, unless we write to the memory. If we want to export the function for other modules, we must put getter prototype outside the implementation into the interface and emplace its contract together with it. Then, other programmers are capable to write its own annotated code that utilize the getter. To ensure that getter returns always the same value, we need to include our private static variable in its contract, because it determines the state of the module. Unfortunately, other code does not see internal data of the module, because it is not defined in the interface.

The solution is straightforward: we must define a concrete state in the implementation (where it already is) and define a ghost structure in the interface that points to it. The interface itself must be able to refer to the internal state of the module. We declare a ghost reference that points to appropriate data memory. This reference may now be used in the function prototype contract in header, because it is not part of the physical code (it is ghost). Programmers cannot change it, therefore it does not need to be hidden to users of the interface. Note that we actually publish the internal state, but only as ghost code and if an analyst using the interface wants to change the state directly with a ghost code, he could not, because the verifier would forbid change of concrete memory from ghost annotation. The final step is to associate the ghost reference with physical data. We

may just add an axiom that assign the address of data to our reference and that is it. Here is a simple example:

```
/* Interface */
_(pure int *interface_value)

int get_value(void)
    _(maintains \thread_local(interface_value))
    _(returns *interface_value)
    _(decreases 0);

void set_value(int param)
    _(writes interface_value)
    _(ensures *interface_value == param)
    _(decreases 0);

/* Implementation */
static int implementation_value = 0;
_(axiom interface_value == &implementation_value)

int get_value(void) { return implementation_value; }
void set_value(int param) { implementation_value = param; }
```

Source code 4.1: One solution to export private state into interface as ghost data

Device state

It looks easy, but the state is usually more complicated than just one primitive variable, thou there may be used structure type as well.

We define three new compound types that we can see in the code 4.2. The DISK_CONTEXT ghost type represents the state of the all disk hardware. It contains only one variable, an array of objects with type DISK_VOLUME. Every object of array is dedicated to one volume. The DISK_VOLUME type contains also one variable volume, a pointer to concrete structure DVOLUME with internal data. DISK_VOLUME also owns the concrete object, so if it is closed, the DVOLUME object is closed too.

DISK_VOLUME as an intermediate type seems to be useless, but it is necessary. We cannot declare just an array of DVOLUME pointers, because we would need an access to pointer variables too (object of pointer type, e.g. &arr[5], not an object of DVOLUME type, e.g. arr[5]). In this case, the pointer is a primitive type and array of pointers is an object that surrounds all the pointers. Therefore we cannot wrap only one element. We need an array of objects, because every object stored in static storage has address that is constantly known at compile time and thou cannot be changed.

Finally, we must inter-connect the implementation with the interface. In the disk module skeleton, we define static array of DVOLUME structures, that represents private data, and introduce the axiom that sets the address of every concrete object to proper pointers in DISK_VOLUME. Obviously, every implementation of the disk module may represent internal data in a other way and it may by difficult to relate ghost objects in the interface with the concrete internal state. This is just one example. Anyway, the solution of this connection is a bit tricky.

```
/* Interface */
_(ghost typedef _(claimable) struct {
    DVOLUME* volume = 0;

    _(invariant \mine(\this->volume))
} DISK_VOLUME)

_(ghost typedef struct {
    DISK_VOLUME volumes[_VOLUMES];
} DISK_CONTEXT)

_(ghost DISK_CONTEXT disk_context)
```

Source code 4.2: Global ghost variable DISK_CONTEXT is array of independent DISK_VOLUME objects that represent interface state and each of them may be wrapped separately

Unreliable storage medium

One can consider the content of a disk as part of the internal state. It may be helpful, for instance, when we would need to perform termination analysis of file cluster chain traversing, we could look directly to the device. The disk abstraction would have to be a type with invariants. However, such a representation of the disk can be tricky. It may seem that the data do not carry real memory fields, but a series of predicates, so they do not take up any space, but the predicates with concrete data take up even more space. The model could be possibly too large and complicated; we actually talk about disk simulator. Moreover, if we wanted to verify disk driver implementation too, we would have no change to mirror the ghost state with the real content of the disk because obviously it is not part of the memory.

Only the functions disk_read and disk_write access the content of disks. If we represent the disk as part of the state, we may interconnect these two functions so that data written and read again are the same. It simulates reliable storage medium and it is one of the ultimate goals of the file system verification. Unfortunately, it is not easy in VCC. We can achieve it only with axiom statement, which is the only mechanism to relate results from two functions. Since these two functions are not pure, we cannot mention it in the axiom statement. Anyway, we do not need it because we consider that **the medium is NOT reliable**. In this case, the file system must check consistency of every read value. It is an interesting task for future work.

Concurrent access to device

A device is by nature a shared medium. It means that several threads may access it. If a function does not change the internal state of the device, it should be possible to call it in more threads concurrently. So, we must ensure that the internal state is closed during the call. This is a classic case where it is appropriate to use claims. We have already prepared DISK_VOLUME structure, which is ideal for claiming. We must annotate it as _(claimable).

Every DISK_VOLUME array entry of disk_context may be owned and claimed separately. A function that only reads the state just need a valid, wrapped claim

as an additional parameter that claims a pointer to DISK_VOLUME. Then, we are sure that no one other changes the disk state when any thread is inside the function. The only exception is disk_initialize function. Since it changes the device state, it requires write access to the wrapped DISK_VOLUME object without any claims. It also requires that the device is exclusively owned by the current thread.

If the device status is claimed, it is also closed. We may safely access to non-volatile fields of DISK_VOLUME, because they never change. However, VCC needs to cast a field with special annotation by_claim. It creates some assertions that checks whether the object of field is really closed. Then, we may read the field [1]. Annotation by_claim is heavily used in file system code because we usually make reasoning about the device state. The concrete code that uses disk interface, may access disk context only trough interface, not directly. However, ghost code and function contract may mention a field of DVOLUME, so we need to cast the field by by_claim somewhere in function body, otherwise the verification fails, even if the function does not call any disk API function.

Function contracts

The interface has exhausting documentation, so is not difficult to write the annotation. The interface should have no parameter requirements because programmer using the interface can pass anything. The functions have only precondition that permits some kind of memory access: A claim is required to access the disk context and blocks of memory must either be thread local (disk_write) or writable (disk_write and disk_ioctl)

Parameter buff of disk_ioctl function has the type of void pointer and the type of data depends on the parameter cmd. For every command that wants to write into the buffer, we must provide a write permission to the function for the proper type. It is also a kind of precondition because VCC permits to write not to an address, but to a typed address, although it is allowed in C to write to an address of arbitrary type (the reasons are described in section 2.2).

In the postconditions, we refer to previously defined device state. For instance, disk_status function returns status of a device that is part of the ghost device state declaration. An error return value of function disk_read, disk_write and disk_ioctl may occur when parameters are not proper, otherwise function succeeds or fails with error of device. There is also added new function disk_volumes_initialize that shows how the internal storage may be initialized and then wrapped at the start of the program.

4.2.2 Objects in FatFs

FatFs defines three essential objects: FATFS, FIL and DIR. FATFS is the main file system object, FIL represents an open file and DIR represents an open directory [2]. In addition to Device Control interface, FatFs module has its internal state. This state is defined in the module private work area in `ff.c` file.

[1] Actually, _(by_claim) is a bit magical. It is described in manual what it should do, but only this annotation works.

[2] It is tedious to constantly talk about files and directories separately. Both of them are objects of file system, but if it is obvious, I shall call them simply as files.

The most important part of the state (and the only one in our configuration) is a list of slots for each logical drive that is represented by a static array FatFs[_VOLUMES] in the implementation. Each slot holds a pointer to a registered FATFS object or it is NULL. Registered file system object may or may not be mounted, i.e. to be a valid object that represents an initialized device with FAT file system. There is a macro is_mounted that simply checks whether a field fs_type is zero or not. A lot of invariants do not need to hold if the object is not mounted.

The first thing we need to define is a consistent state of the module. This is such a state that exists before and after a call of an API function. Every API function must maintain the state, even if it is not consistent inside the call. State of the module is consistent if every logical drive is either undefined (NULL) or a FATFS object is registered. Every registered object must be in a consistent state. VCC naturally defines a consistent state of an object such that it is closed. Unlike Device Control interface, we additionally require that the registered object is wrapped, i.e. owned by a thread, which calls a API function, for several reasons:

- Most of the API functions write into FATFS object (e.g. because of delays mounting), so even if FATFS type was claimable, we would have needed the object to be wrapped.
- Although we do not verify locks of FatFs, the application code can have its own locks and thus it solves this limitation.
- Wrapped object is easier to verify. Concurrent access to the file system is much more difficult to verify and we are not even trying to achieve this. We only show that each logical drive can be serviced by another thread.

We demonstrate what invariants must hold in FATFS object to be consistent and to represent FAT file system stored on a disk. We show that the FATFS object and all open files belong to one file system, to one tree of ownership. At first, we describe how a consistent FATFS object looks like without files, which is a bit simpler abstraction, and then we show all invariants of tree of ownership with files.

File system module state

It is a similar approach like in case of the disk context. We must export the state of the file system module. There is a new type, FS_CONTEXT that holds a pointer into the internal array of FATFS pointers. We define a global variable fs_context with field fatfs that is mapped by an axiom to FatFs[_VOLUMES]. Then, for every i < _VOLUMES, every field at fs_context.fatfs[i] is always equal to the field at FatFs[i].

Because of inconsistency in VCC, we cannot wrap one element place of the FatFs static array separately, it must be mutable. It means that also the whole FatFs array must be mutable. Since we do not want to put it to the ownership of another object in our project, it does not make any problem. However, it may be a problem for further verification.

FATFS as the owner of device

Intuitively, each registered FATFS object has one related physical device. Only this object should access the device and also change it. The best way to achieve this is to "own" the device, i.e. include state &disk_context.volumes[vol] into object's tree of ownership, where vol is the number of volume. Once FATFS object is opened, we have exclusive access to the device. The main advantage of the ownership is that we can mention the state of device in file system object's invariants. Together with the device, we add dc field that is claim to the device. It allows us to call functions of Device Control interface. Since we own the devices, we do not need to keep the claim, because we can always create it, but it is unnecessary extra code.

There is added new field drive in FATFS that indicates number of physical drive. The difference between the drv field is such that this field represents physical drive that is owned by the file system. The device is owned even if the object is not mounted. It is enough when the object is stored in static storage. If the file system object is mounted, these two fields are equal. There are other two invariants over device that must hold when the object is mounted. The device must be initialized, obviously, and sector size in the file system object must be equal to sector size of the medium.

Invariants of FATFS

The rest of the invariants define the structure of FAT file system. If file system object is not mounted, there is no valid FAT file system and invariants make no sense. They must hold only if field fs_type identifies one of three FAT sub-type. All these "structural" invariants are presented as implication and the premises is always macro is_mounted. There are many invariants and most of them are evident, e.g. n_fats == 1u || n_fats == 2u. We present only some more complicated invariants.

```
fs_type == fs_type_from_clusters(n_fatent - 2u)
```

FAT sub-type depends precisely on number of clusters. fs_type_from_clusters logic macro just checks whether number of clusters is less than 4086 constant (FAT12), or less than 65526 (FAT16) or greater (FAT32)[3].

```
fsize * ssize >= ((fs_type == FS_FAT12)
    ? (n_fatent * 3u / 2u + (n_fatent & 1u))
    : ((fs_type == FS_FAT16) ? n_fatent * 2u : n_fatent * 4u))
```

FAT table is large enough for all cluster entries. The size of the table depends also on FAT sub-type, because every type can contain different maximum of clusters. When we read Boot sector, number of sector in a FAT and number of clusters are independent values, so the invariant is may be simply violate.

```
database == fatbase + n_fats * fsize
    + n_rootdir / (ssize / SZ_DIR)
```

[3]Precise determination of these constants is not set by the FAT architecture, but rather historically. It differs in many implementations and it creates inconsistencies. Therefore, the file system should not have the number of clusters close to these values.

The is precisely defined number of sectors between reserved region and data region. There are 1 or 2 FAT copies and root directory. The value ssize / SZ_DIR is number of directory entries that fits into one sector, where SZ_DIR is size of one entry, so the root directory must be large enough to contain all its entries.

```
((fs_type == FS_FAT32) ? database : dirbase)
    == fatbase + n_fats * fsize
```

Since there is no root directory in FAT32, there is only FAT table between reserved region and data region. Otherwise, dirbase denotes for the first sector of root directory that is placed after FAT table.

```
fs_type == FS_FAT32 ==> (dirbase >= 2u && dirbase < n_fatent)
```

In FAT32, the root directory can be stored in any valid data cluster. The root directory is important for traversing a full path, so it is useful to keep it. Field dirbase must be a valid cluster.

```
database + (n_fatent - 2u) * csize - 1u <= (DWORD)-1
```

We must show that all sectors are located on the medium. The last address that FAT file system may recognize has number 0xFFFFFFFF (that is the same as constant (DWORD)−1, the maximum value of 4-byte type DWORD). So, there cannot be a sector with higher number. The last sector of FAT partition is the last sector of the last data cluster. The last sector is usually not used, because the driver code may overflow when it handles the number of sector. We have shown that it cannot happen in FatFs.

Ownership of objects `FIL` and `DIR` in `FATFS`

We extend FATFS ownership with FIL and DIR objects. File system logically contains files and directories. However, it makes a dilemma. If we say that the file system object owns all open file objects, we will not be able to wrap the files. Every file object has some invariants, among others the invariants refer to the file system objects which are part of. If we want to refer to this object and wrap the files at the same time, we need the FATFS object to be wrapped too because invariants cannot reason about a field of a structure that does not own. However, the file cannot own file system object, for logical and even practical reasons because two files cannot own one FATFS object.

The solution is not clear the first time. It is evident that both file system object and file objects must hold their invariants together. So, the solution is to push all invariants from FIL and DIR into the FATFS. First, all files belonging to file system must be wrapped and put into \owns. We may wrap the files now because there are no invariants referring to the FATFS object. Second, all invariant must hold. The invariants moved from FIL and DIR specify the structure of file systems and also all files. Finally, the file system object may be wrapped. There are other solutions, for instance, create a ghost object that would own all the objects. However, our solution seems to be simpler.

We must somehow identify that a file belongs to file system or not. The nice solution is to create map from file structure to boolean that say whether appropriate

object is included or not. Invariant with ownership must be very simple. However, an invariant with equivalence is not so simple to be admissible. The automatic handling with \owns set cannot be guaranteed. We may disable it and handle the \owns set manually. We must annotate the structure with _(dynamic_owns). In addition, we add three invariants:

- The ownership is related to map. Since the \mine annotation is not used as the outermost function of a top-level conjunct of an invariant, we must annotate the FATFS type with _(dynamic_owns)
- If a file system has no valid FAT type, it cannot contain any open files. It allows us to check that if we are about to invalidate FATFS object, we must invalidate all open files at first.
- We must specify the link between the file and the file system: fil->fs == fs && fil->id == fs->id.

The function code in module must now set explicitly \owns set of object, if it needs to wrap/unwrap. We must precisely set that it contains the device object and claim on it. The map may be set with an expression with lambda quantifier.

4.2.3 Invariants of `DIR` and `FIL`

In case of file and directory type, we must distinguish invariants that link to members of file system object and those that do not. The first group is usually simpler. These invariants are associated with the FIL or DIR type directly, so they must hold when the object is wrapped.

Other group represents mostly more complicated conditions because they have access to file system data. They define the consistent state of file indirectly because these invariants are declared as part of file system object. Since the FATFS may contain multiple files and the invariants must hold for all of them, we must disable automatic ownership management by defining of _(dynamic_owns) annotation (as we have already done) and put equivalence that relates ownership with these invariants.

Generally, it is much desirable to put as many invariants as possible into file object than into file system object because the file system object has already many invariants and other worsen the verification. When unwrapping FATFS, we may not need to know anything about files and so these invariants are not instantiated. Simply said, the verifier does not need to know them when it operates only with FATFS.

The first group of invariants is not so interesting; the invariants show only obvious facts. The FIL structure has not such interesting even the invariants referring to FATFS. However, invariants of DIR are more interesting. There are those that determine current cluster, current sector or position in sector where the current directory entry is. Because of many operations of non-linear arithmetic, it is extremely difficult to prove them, for both the VCC and users.

```
index < fs->ssize / SZ_DIR * fs->csize
    ==> clust == ((fs->fs_type != FS_FAT32 || sclust != 0u)
    ? sclust : fs->dirbase)
```

If we get an index and start cluster, we are able to determine the current cluster only when we traverse the cluster chain of the directory. It is a property that we cannot express in an invariant. However, we are able to show when the entry is still located in the start cluster. If the current index is less than number of cluster entries, the current cluster is the first cluster. The fs->ssize / SZ_DIR is number of entries in sector and therefore fs->ssize / SZ_DIR * fs->csize is the number of entries in the cluster.

```
(clust != 0u) ? (sect == fs->database
    + (clust - 2u) * fs->csize + (DWORD)index
    % (fs->ssize / SZ_DIR * fs->csize) / (SS(fs) / SZ_DIR))
    && sect == fs->dirbase + index / (fs->ssize / SZ_DIR))
```

Resolving the current sector is the one of the most difficult thing to prove in the project. We must distinguish current sector in root and ordinary directory. For the root directory, the local number of the current sector is value index / (fs->ssize / SZ_DIR). For the ordinary directory, it is much harder computation. First, we must find the number of the first sector of the current cluster. Second, we must calculate local index relative to the current cluster (DWORD)index % (fs->ssize / SZ_DIR * fs->csize. If we divide it with number of entries per cluster, we have sector number relative to the current cluster. With the first sector of the current cluster, we have the current sector

```
dir == fs->win + index % (fs->csize / SZ_DIR) * SZ_DIR
```

The current directory entry is stored in disk access window. The expression index % (fs->csize / SZ_DIR) denotes for index of entry in the current sector. Entries are aligned to 32 bytes (SZ_DIR), so we may compute the pointer to the current entry easily.

4.3 Non-linear arithmetic proofs

We talk about non-linear arithmetic if its formulae use multiplication, division or modulo operations. The decision problem for non-linear integer arithmetic is undecidable in general and VCC (or rather Z3) has no effective method to find an integral solution. The behavior of the solver is then very unstable. The verification process may finish immediately or run forever because the solver tries to decide a task equivalent to Halting problem.

VCC is able to prove fairly conditions with addition, subtraction and also multiplication in most cases. For division, modulo and some properties of multiplication, it is absolutely indispensable to define or write suitable arithmetic rules in VCC as ghost code.

Every rule is defined as a ghost function that represents it in this way:

- function parameters are variables bounded by universal quantification
- function preconditions are initial conditions of the rule
- function postconditions are the conclusion of the rule

Moreover, rules are not only functions prototypes, every rule is proved with annotated C code. These proofs do not contain any unnecessary assertion in their function bodies except those with quantifier and dead code detection.

Functions have many advantages over axioms and asserts with quantifiers. VCC generally deals with quantified equations harder than with those with free variables and more bound variables there are, more difficult it is (even for three variables, the verifier often finds no solution). Moreover, annotation cannot utilize more complex structures like loops nor other already proven rules.

Axioms has the disadvantage that must be defined outside functions in scope of the entire verification unit (a file). Thus, even functions that do not explicitly need the facts would have to include them into reasoning. Even asserts with quantifiers are slower because the verifier must deduce relations between concrete variables first. Functions call proves the fact just for expressions passed as parameters that are perfectly the things that functions need. All of these things reduces verification condition, reduces need to prove non-linear arithmetic and accelerates verification as the result.

Although the file system itself does not need to handle non-linear operations with negative numbers, \integer type is the preferred type of parameters, instead of \natural type (that is its subset) basically for two reasons. First, VCC is often not able to prove a rule for \natural parameters, although for \integer parameters it is. And second, it is often more clear to show the rule for all integers.

The fundamental proof is implemented in divide_integer function. There is the trivial algorithm of Euclidean division, based on repeated subtraction (we do not need to deal with its inefficiency, we just want it to be correct). The algorithm returns quotient (q) and remained (r) as a result, that are unique for every pair of numerator (n) and denominator (d). This algorithm gives the same result as division and modulo operations implemented in C language (and so understood even by VCC and Z3). This is not possible to prove, we must assume the equation $n / d = q$ in the language as a fact, where modulo is derived from division by formula $n = n / d * d + n \% d$. Now all predicates proved for quotient and remainder are also proved for division and modulo operations. There are almost no other assumption and all properties are proven due to assertions with universal quantifier or by using other already proven rules.

The most common method of mathematical proof is mathematical induction, achieved by loops or recursion. It has disadvantage that we may prove only the mathematical induction with step that decreases numerator or denominator because these two may determine the variant of the loop or recursion that is decreasing only if these parameters are decreasing. Because of this, rules $n / d = (n + d)/ d - 1$ and $n = (n * d) / d$ are assumed to be sound. Another type of variant reveals in functions that prove rules also for negative values. A function usually proves a rule for only positives parameters, otherwise it negates them and calls itself recursively. Variant of such function is an expression that gives a larger value for negative parameters.

The other useful method of proving that we call identifying of zero quotient is implemented in identify_zero_quotient function. It simply say that if result of multiplication $a * b = c$ is less than one of the coefficient (and greater then negative value of the same coefficient for integers), the result is zero and the other coefficient is also zero. It exploits the fact that remainder is always bounded by the

denominator. Let say that b is the denominator and c is a remainder after dividing of a number by b or difference of two remainders. This is a scenario where we divide one equation from other and a is a difference of two expressions. The rule indicates that these two expressions are equal. The mechanism usually shows relation between result of two pairs of numerator and denominator (e.g. how related are results of pair (n, d) and pair $(n - d, d)$).

5. Related works

Since this work is aimed on verification of file system, it is worth noting other efforts both to create a verification tool for C language and also to verify a file system. Unlike attempts to verify a concrete file system, which are rare, there are several contract-based tools for C language.

5.1 Automatic verification of C programs

Framework for Modular Analysis of C programs (known better as *Frama-C*)[9] is a suite of analyzers and plug-ins designed for static analysis of source code in C language. A single static analyzer may work alone or employ the results of other plug-ins in a collaborative framework, which makes the tool very strong. Frama-C allows to analyze source code with formal specification that can be written in the ANSI/ISO C Specification Language (ACSL). ACSL is a language for C programs that uses classic design-by-contract paradigm with preconditions, postconditions, invariants, etc. Specification is embedded in source code as annotation in C comments, similar to Java Modeling Language (JML).

There are two plug-ins incorporated into Frama-C that performs deductive verification of C programs with ACSL, Jessie and Wp. *Jessie* is a plug-in that converts an annotated C program to its Jessie intermediate language. The language is part of Why[32] platform that is used as back-end for Jessie plug-in. It is a general-purpose verification condition generator that extensively uses external automatic provers. The *WP* plug-in refers to itself as a novel implementation of a Weakest Precondition calculus for annotated C programs, which focuses on parameterization with respect to memory model. It takes different, more low-level approach than Jessie.

VeriFast [15] is a verification tool for verifying correctness properties of single-threaded and multi-threaded C and Java programs. The verifier is able to read notation in a separation logic. The specification language allows to use advanced constructs like inductive data types, primitive recursive pure functions over these data types, abstract separation logic predicates, fixpoint functions, generics, etc. These construct may exist only like a ghost code and user may write lemma functions. Since version 7.0, it also supports verifying of full functional partial correctness of lock-free data structures. VeriFast uses Z3 the underlying SMT solver.

The VCC tool has become the base verifier for *VCDryad* [23], an automated deductive framework for C programs. VCDryad extends the VCC and it is a synthesis of the VCC and Dryad, a dialect of separation logic with recursive definitions. Separation logic of Dryad (and thus also VCDryad) is based on natural proofs. Natural proofs represents proof tactics that enable automated reasoning exploiting recursion, which looks more like human proofs. Dryad provides natural proofs for properties of structures, data, and separation. Unlike other tools, VCDryad does not require any additional proof tactics other than function contracts and loop invariants that advise underlying prover how to proceed.

Escher C Verifier (*eCv*)[12] is a tool for developing verified C code for safety-critical software systems. Its developer is Escher Technologies. It shares ma-

ture automated theorem prover with other company product, Perfect Developer. Verifier support only a subset the C language without unsafe features (based on MISRA-C 2012). On the other hand, it is stated that the prover is able to prove nearly all programs without user intervention.

5.2 Analysis of file system

One of the most promising projects has appeared in NICTA, Australia's Information Communications Technology (ICT) Research Centre. Here, they attempt to develop trustworthy, fully verified file system[1]. Most of previous attempts to verify a file system used model-checking paradigm, but this project uses verification methods for functional correctness. They have chosen BilbyFS, a high-performance flash file system, for this goal. It is designed to be highly modular. Single components are represented in a set of domain-specific languages. Then, design-level specification and its C implementation is produces from them. Implementation details are introduced only when they refine single components.

Authors of the paper [2] presents correctness proof of a basic file system implementation. They choose standard Unix file system architecture for their implementation where are such object as inodes and fixed-size disk blocks. Their specification is formalized as map from filenames to sequences of single bytes. The point of correctness verification is to prove the existence of a simulation relation between the specification and the implementation. The proof is expressed and then checked with Athena, a programming language and interactive theorem-proving framework based on denotational proof languages.

In the paper [10], do not provide an implementation of a file system, but they decided to verify a public file system interface instead. They choose POSIX standard and wrote a formal specification of some POSIX standard functions using the modeling language VDM++. The verification progress is divided to verification of object and then public function.

The last paper [17] shows the analysis and formal modeling of a flash-based file system in Alloy language. Alloy is a language for describing structures, but also model solver takes the constraints of a model. The presented model addresses three main aspects of a flash file system:

- The underlying flash hardware.
- The file system software with principal file operations, such as read and write.
- A fault-tolerance mechanism for handling unexpected hardware failures.

The model also includes techniques for efficiently managing block erasures, such as wear-leveling and erase-unit reclamation because flash memories have upper limitation to number of write operations.

Conclusion

In this thesis, we have presented the possibilities of verification of the real file system. The first goal was to choose the file system appropriate for verification. The classical FAT file system is simple and therefore suitable for verification, yet practical that its analysis may be helpful. The FatFs module that we have chosen has the advantage that is highly configurable and it may be reduced to a very light-weight system.

The next goal was to provide a specification for a non-trivial part of the file system. We have designed invariants for three fundamental types of the module and function contract for about 20 functions and several auxiliary functions and macros.

The last goal was to analyze our annotated code with VCC tool. Most of function and all type definitions are verified and several function verified too, but their verification is unstable, i.e. it may or may not terminate. The VCC has proven itself as very powerful verifier that is able to handle the complexity of C language. However, sometimes it is difficult to understand its behavior. One of the biggest problems of VCC has limited support for non-linear arithmetic. For this reason, the project contains the little library of basic non-linear arithmetic rules, including proofs in VCC itself.

There is still a room for improvement of the specification. The FatFs module has many features that are not yet analyzed, e.g. locks. Another challenge is to simulate reliable storage, which may allow verification of deeper properties of the FAT system itself. For instance, we are not able to prove that traversing of cluster chain terminates.

Bibliography

[1] Sidney Amani, Leonid Ryzhyk, Toby Murray: *Towards a Fully Verified File System*, Poster presentation at EuroSys Doctoral Workshop, Bern, Switzerland, April 2012

[2] Konstantine Arkoudas, Karen Zee, Viktor Kuncak, Martin Rinard: *Verifying a file system implementation*, 6th ICFEM, 373–390, Seattle, USA, November 2004

[3] Christoph Baumann, Bernhard Beckert, Holger Blasum, and Thorsten Bormer. *Ingredients of operating system correctness*, In Embedded World 2010 Conference, Nuremberg, Germany, March 2010

[4] Alfs T. Berztiss, Mark A. Ardis: *Formal Verification of Programs*, SEI Curriculum Module SEI-CM-20-1.0, December 1988

[5] ChaN: *FatFs —Generic FAT File System Module.* [Online] `http://elm-chan.org/fsw/ff/00index_e.html`

[6] Chishm: *libfat: A FAT library for the Nintendo GBA and DS.* [Online] `http://chishm.drunkencoders.com/libfat/`

[7] *VCC CodePlex home page.* [Online] `http://vcc.codeplex.com/`

[8] Ernie Cohen, Markus Dahlweid, Mark A. Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, Stephan Tobies: *VCC: A Practical System for Verifying Concurrent C*, Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Springer, 2009

[9] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, Boris Yakobowski: *Frama-C: A Software Analysis Perspective*, Proceedings of the 10th International Conference on Software Engineering and Formal Methods, SEFM'12, 233–247, Berlin 2012

[10] Simao Cunha, Luís Rodrigues, Augusto Silva, Rui Gonçalves, Samuel Silva: *Verified File-System v1.0*, Department of Informatics - University of Minho, Braga, September 27, 2007 —`http://wiki.di.uminho.pt/twiki/pub/Research/VFS/WebReferences/vfs_my_V1.0.vpp.pdf`

[11] *Volume and File Structure of Disk Cartridges for Information Interchange.* Standard ECMA-107 (2nd edition), ECMA, June 1995

[12] Escher Technologies: *Escher C Verifier.* [Online] `http://www.eschertech.com/products/ecv.php`

[13] *FullFAT: High Performance Embedded FAT 12/16/32 Driver with LFN Suport.* [Online] `https://code.google.com/p/fullfat/`

[14] Mike Gordon: *Background reading on Hoare Logic*, 2012

[15] Bart Jacobs, Frank Piessens: *The VeriFast program verifier*, Technical Report CW-520, Department of Computer Science, Katholieke Universiteit Leuven, Belgium, August 2008

[16] Rajeev Joshi, Gerard J. Holzmann: *A Mini-Challenge: Build a Verifiable Filesystem*, Formal Aspects of Computing 19, 269–272, June 2007

[17] Eunsuk Kang, Daniel Jackson: *Formal Modeling and Analysis of a Flash Filesystem in Alloy*, Proceedings of the 1st International Conference on Abstract State Machines, B and Z, ABZ '08, 294–308, London, UK 2008

[18] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, Simon Winwood: *seL4: Formal verification of an OS kernel*, ACM Symposium on Operating Systems Principles, 207–220, Big Sky, MT, USA, October 2009

[19] Freescale Semiconductor, Inc.: *Freescale MQX™File System (MFS).* [Online] `http://www.freescale.com/webapp/sps/site/overview.jsp?code=MQXMFS`

[20] Microsemi Corporation: *SmartFusion® cSoC: Implementation of FatFs on Serial Flash*, Application Note AC360, Revision 4, January 2013

[21] *Microsoft Extensible Firmware Initiative FAT32 File System Specification, FAT: General Overview of On-Disk Format*, Version 1.03, Microsoft, December 6, 2000

[22] NASA Langley Formal Methods Site: *What is Formal Methods?*, NASA Langley Research Center [Online] `http://shemesh.larc.nasa.gov/fm/fm-what.html`

[23] Edgar Pek, Xiaokang Qiu, P. Madhusudan: *Natural Proofs for Data Structure Manipulation in C Using Separation Logic*, PLDI '14 Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation , 440–451, ACM, 2014

[24] Sandip Ray: *Using Theorem Proving and Algorithmic Decision Procedures for Large-Scale System Verification*, December, 2005

[25] Renesas Electronics Corporation: *FM3S-TFAT-Tiny.* [Online] `http://www.renesas.com/products/tools/middleware_and_drivers/tiny_soft/tfat/m3s_tfat_tiny/index.jsp`

[26] Yaniv Sa'ar: *Deductive and Algorithmic Methods for Formal Verification*, October 30, 2007

[27] Alok Sanghavi: *What is formal verification?*, EE Times-Asia, 2010

[28] Kenneth Slonneger, Barry Kurtz: *Formal Syntax and Semantics of Programming Languages: A Laboratory Based Approach*, 1st Edition, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. [Online] `http://homepage.cs.uiowa.edu/~slonnegr/plf/Book/`

[29] *The VCC Manual*, Working draft, version 0.2, August 20, 2012. [Online] `http://vcc.codeplex.com/`

[30] Michał Moskal, Wolfram Schulte, Ernie Cohen, Mark A. Hillebrand, Stephan Tobies: *Verifying C Programs: A VCC Tutorial*, Working draft, version 0.2, August 25, 2012. [Online] `http://vcc.codeplex.com/`

[31] Verisoft XT: *The Verisoft XT project*. [Online] `http://www.verisoftxt.de`

[32] *Why: a software verification platform*. [Online] `http://why.lri.fr/`

A. Content of CD-ROM

The master thesis copy in PDF format is located in root directory with the name `Master_Thesis_-_David_Skorvaga.pdf`. There are also three directories described in table A.1.

Directory	Description
`install`	VCC Installer and installation packages of frameworks that VCC needs.
`original`	Package with original FatFs R0.10b (without annotation)
`project`	The annotated FatFs file system

Table A.1: Directories in root directory of CD-ROM

All the files of FatFs, including files with annotation, and additional scripts are described in the table A.1.

File	Description
`arch/empty/diskio.c`	Dummy implementation of Device Control interface, used to test specification of the interface.
`arch/empty/ffconf.f`	Configuration of FatFs used during the verification.
`src/arithm.{c|h}`	Library with non-linear arithmetic rules and their proofs.
`src/diskio.h`	Annotated Device Control interface.
`src/ff.h`	Annotated Application interface with object invariants of FATFS, FIL and DIR.
`src/ff.c`	The main file of FatFs with the implementation and all the annotation
`src/integer.h`	Definitions of basic data types
`tools/analyze.cmd`	Script that performs inspection by VCC
`tools/combine.cmd`	Complex script that performs automatic analysis for multiple FatFs configurations
`tools/compile.cmd`	It compiles FatFs with MS Visual Studio
`tools/verify.cmd`	Script that verifies the annotated part of file system, including objects
`tools/verify_arithm.cmd`	Script that verifies non-linear arithmetic
`tools/verify_diskio.cmd`	Script that verifies dummy Disk interface
`tools/verify_function.cmd`	Script that verifies a single function
`fatfs.{sln|vcxproj}`	MS Visual Studio project for compilation of FatFs source code
`Makefile`	Makefile for Unix system

Table A.2: Files of the project (mainly annotated code of FatFs)

B. Installation instructions and running the verification

To verify the included source code, we used no more application than command line tool. There are some prerequisites needed to successfully run VCC. All packages can be found on VCC web pages[7] and they should be installed the usual way. There is a list:

- Microsoft .NET Framework v4.0 or later
- Microsoft Visual Studio 2010 F# Runtime 2.0
- The Old F# "PowerPack"
- Microsoft Visual C++ 2010 Redistributable Package (x86)
- VCC Installer v2.3.10214.0, Feb 14, 2013

After the successful installation, `vcc` is added. The command takes files with C language source code as parameters and performs verification. There are some useful options:

- `/functions:<function list>` – List of functions that VCC should verify. If a function calls other ones, only their contract is needed.
- `/warn:<n>` – Set warning level (0-2). Every hind during verification helps.
- `/termination:<n>` – It performs termination analysis. There are four levels, 0 does not analysis, level 3 is the deepest analysis.
- `/smoke` – Run smoke tests that checks dead code. **Caution!** This slows the verification significantly.
- `/z3:/rs:N` – It executes verification with different seed N. Useful for unstable functions. In general, option `/z3` pass on option to Z3.
- `/b:/restartProver` – It performs restarts verification for every function separately. In general, option `/b` pass on option to Boogie.

To perform the entire project verification, call script `verify.cmd`.

The verification takes some times. It was executed on the computer with processor Intel® Core™ i3-2310M with frequency 2.10 GHz and verification was running approximately 45 minutes. All options `/warn`, `/termination` and `/smoke` are set. The script does not perform verification of functions that are marked as unstable. However, there is `verify_function.cmd` script that performs verification for a function with name sent as parameter. Furthermore, `verify_diskio.cmd` script perform analysis of disk interface with a dummy implementation and `verify_arithm.cmd` script proves auxiliary arithmetic rules.

C. Found bugs

Function	Line	Bug description
sync_window	976	Return value of disk_write is not check, possible inconsistent between
remove_chain	1782	Higher values than number of entries are not treated as if they were end-of-chain markers
create_name	3904	File name that has only extension is not rejected.
check_path_component	4079	Start cluster number from disk is not checked.
find_partition	4594	Function can mistakenly identify an invalid sector as FAT Boot sector
initialize_fs_basics	4669	Number of sectors per FAT can be 0
initialize_fs_basics	4680	Size of FAT copies can overflow
initialize_fs_type_and_limits	4765	Size of reserved region + FAT copies can overflow
initialize_fs_type_and_limits	4769	Size of reserved region + FAT copies + root directory can overflow
initialize_fs_type_and_limits	4790	Some sectors can be inaccessible
initialize_fs_type_and_limits	4793	Number of clusters can be greater than FAT32 limit
initialize_fs_fat_table	4851	Start cluster number of root directory in FAT32 is not checked
initialize_fs_info	4949	If FSINFO is enabled, there is no check that there are at least two reserved sectors
f_open	5860	Directory entry can point to incorrect start cluster
f_open	5863	There can possibly be a start cluster when file has no size

Table C.1: List of bugs that were found by the verification, with precise direction to the `ff.c` file

List of Tables

List of Figures