TUHH
Technische Universität Hamburg-Harburg

INSTITUTE FOR SOFTWARE SYSTEMS

Project Thesis

(Studienarbeit)

Alexander Galkin

IIW Diploma Student
Matriculation ID: 34127

Inferring Alias Contracts in VCC using Separation Analysis

Supervisor: Prof. Dr. Sibylle Schupp

May 2011

Erklärung

Hiermit erkläre ich, dass ich die von mir am heutigen Tage dem Institut für Software Systems eingereichte Studienarbeit vollkommen selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Alle Abbildungen in dieser Arbeit sind eigene Darstellungen. Wo Abbildungen übernommen und abgeändert wurden, ist die Quelle der ursprünglichen Abbildung angegeben.

Hamburg,

. . . .
(Unterschrift)

Acknowledgments

Before all, I would like to thank to my family, first of all my wife who made me believe I can become a Software Engineer and who was supporting me all the time during my study.

I would like to thank to my supervisor patience, both generous and prodding attitude to my work, personal supervision of my thesis and last but not least for her immense efforts to help me improve my English writing skills.

I would like to thank Michał Moskal for his constructive criticisms and suggestions during our meeting in Seattle, Sacha Böhme for his prompt and detailed answers to my numerous questions, Claude Marché for providing me with PhD theses of his graduate students and very helpful explanations, Sven Mattsen for pointing out to the weak points and helping to improve those, and Matko Botincan for information on the status of his project and valuable remarks.

Abstract

The main goal of this work is to enable an interplay between two different verification tool chains: Frama-C, developed concomitantly by *Commissariat à l'Énergie Atomique et aux Énergies Alternatives* and *Inria,* and the Verified C Compiler, developed by *Microsoft Research.* Both tools have a long history and contain powerful algorithms for static verification of C source code.

Both tools address the problem of aliasing, where the memory representation of two or more variables overlap (partial aliasing) or even is shared entirely (full aliasing). Aliasing is a common problem in software verification, for it produces side effects during code analysis by changing the values of variables that were not explicitly (re-)assigned. This problem is apparent in weakly typed languages like C and requires special attention.

The byte-level memory model of VCC is tuned narrowly to the verification of concurrent programs, but this memory model turns out not to be optimal for alias detection and memory safety checks. Frama-C and its plug-in Jessie, on the other hand, implement deductive verification using a region-based memory representation, which is better suited for separation and alias analysis.

We use the memory safety analysis of Jessie to derive missing pre-conditions in our VCC specification that enforce memory safety. To implement the necessary constraints in VCC, we have to confine alias detection to aliases of primitive data types and consider only full aliasing. These assumptions allows us to express the aliasing condition in the form of pointer inequality: two variables are not aliased, if their pointers are unequal.

The main contribution of this thesis is Jessifier, a command-line utility for automated inference of missing alias contracts implemented in C#. This utility uses the outcome of the analysis performed by Jessie to detect possibly missing VCC annotations and attempts to generate them automatically (to infer). This inference is only possible if aliases are directly present in function invocations in the source code. If not, the respective verification conditions turn into tautologies and eliminated by Jessie from the analysis output.

Zusammenfassung

Das Hauptziel dieser Arbeit liegt darin, eine Zusammenarbeit zwischen zwei unterschiedlichen Verifikationswerkzeugketten zu ermöglichen; es wird das Werkzeug Frama-C, einerseits, das gemeinsam von *Commissariat à l'Énergie Atomique et aux Énergies Alternatives* und *Inria*, betrachtet und dem Verified C Compiler entwickelt von *Microsoft Research* gegenübergestellt. Beide Werkzeuge haben eine lange Geschichte und beinhalten mächtige Algorithmen für die statische Verifikation von C-Quellcode.

Beide Tools behandeln auch das Aliasing-Problem, bei dem die Speicherdarstellungen von zwei oder mehreren Variablen sich überschneiden (partielles Aliasing) oder ganz überlappen (vollständiges Aliasing). Aliasing ist ein bekanntes Problem in Software Verifikation: bei der Code-Analyse führt es zu Nebeneffekten, dass Variablen, die nicht explizit geändert wurden, doch neue Werte annehmen. Das Problem tritt besonders in schwach-typisierten Sprachen wie C auf und muss extra beachtet werden.

Das Byte-Level-Speichermodell von VCC ist streng an die Bedürfnisse der Concurrent Verifikation angepasst und für Aliaserkennung und Speichersicherheit nicht optimal. Jessie, ein Plug-In für Frama-C für unterschiedliche Arten der deduktive Verifikation, basiert auf einem Region-Speichermodell und ist daher besser für die Aliasing-Analyse geeignet.

Wir setzen die Speichersicherheitsanalyse von Jessie ein, um fehlende Vorbedingungen in einer VCC-Spezifikation herzuleiten, damit Speichersicherheit gewährleistet wird. Damit wir die nötigen Vorbedingungen in VCC implementieren können, müssen wir uns bei der Aliaserkennung auf primitive Datentypen einschränken und nur das vollständige Aliasing in Betracht ziehen. Diese Annahmen ermöglichen uns, Aliasbedingungen in Form einer Zeigerungleichheit auszudrücken: zwei Variablen sind keine Aliases wenn ihre Zeiger ungleich sind.

Der Hauptbeitrag dieser Arbeit ist der Jessifier, eine Konsolenanwendung für die automatisierte Erkennung von fehlenden Aliasverträgen, der in C# implementiert wurde. Diese Anwendung benutzt die Ausgabe von Jessie, um möglicherweise fehlende VCC-Verträge aufzuspüren und versucht diese automatisiert zu erstellen. Dies ist nur dann möglich, wenn die Aliase direkt in den Funktionsaufrufen im Quellcode vorhanden sind. Sonst werden die entsprechenden Verifikationsbedingungen zu Tautologien und durch Jessie aus der Ausgabe eliminiert.

Contents

1 Introduction **1**

2 Software validation and verification **4**

2.1 Software validation vs. software verification 4
2.2 Approaches to software validation 5
2.3 Specifications and contracts 6
2.4 Deductive verification 8
2.5 Modular verification 9

3 Verification tools and tool chains **13**

3.1 Software verification workflow 13
3.1.1 Outcome: Verification was successful 14
3.1.2 Outcome: Verification fails 14
3.1.3 Outcome: Verification timed out 15
3.2 Software Verification Tool Chain 15
3.2.1 Original C Code 17
3.2.2 Annotating code 18
3.2.3 Annotation inference 20
3.2.4 Conversion from annotated C to intermediate proof languages 20
3.2.5 Generating Verification Conditions (VCs) 22
3.2.6 SMT solvers 22

4 Memory models **24**

4.1 Type-safe memory models 24
4.2 Aliasing 25
4.3 Memory models in VCC and Jessie 25
4.3.1 Explicit specification of the memory model in Jessie 28
4.4 Separation analysis 28

5 Extended Example **30**

5.1 C code samples 30
5.1.1 Code characteristics 31
5.1.2 The max() function 31
5.1.3 Contracts and annotations 32
5.2 Alias detection with Jessie and VCC 33
5.2.1 Separation of memory regions in Jessie 33
5.2.2 Memory separation analysis in the scope of a complete C program . 34
5.2.3 Adapting the example for VCC 35
5.2.4 Modular approach to static source code analysis in VCC 36
5.3 Inferring alias contracts from Jessie and Why 37
5.3.1 Determining the level of abstract for annotation inference 37
5.3.2 Using Why code to infer annotations for VCC 37

5.4 Using inferred annotations to strengthen the specification in VCC 39
5.5 Generalizing the sample code 39
5.6 Automated annotation inference 41
5.6.1 Implemented heuristics 41
5.6.2 Application domain of the inference algorithm 44

6 Discussion and conclusions 45
6.1 Difficulties of inferring an annotation 45
6.2 Related work 46
6.3 Future work 47

A Appendix 48
A.1 Sample C program annotated for Jessie and VCC 48
A.2 Jessie code of the sample C program 49
A.3 Why code of the sample C program (abridged) 50
A.4 The `DoParse()` function of Jessifier 57
A.5 Regular expressions used for code parsing 60

Bibliography 62

List of Figures

2.1 Approaches to software verification 6

3.1 Typical workflow in deductive software verification illustrated by tools around VCC (adapted from [Cohen 09b]) 14
3.2 Comparison of the software verification tool chains of Jessie and VCC . . . 17
3.3 Direct comparison between C annotations and their translations in Jessie . 21

4.1 Types of aliasing: no aliasing (top), complete aliasing (middle), and partial aliasing (bottom) 26
4.2 Byte-level memory model of VCC (top) and byte-level block memory model of Jessie (bottom) 26

5.1 Verification conditions for Listing 5.2 shown in gWhy 33
5.2 Output of gWhy with two assertions inferred by separation analysis (left) and the underlying Why code (right) 38
5.3 Algorithm for automated contract inference for VCC from Why code 43

Listings

2.1 Simple implementation of an abs() function in C . . . 6
2.2 Implementation of the abs() function in C decorated with assert() statements 7
2.3 The abs() function with ACSL annotations . . . 8
2.4 The birthday() function annotated with VCC2 . . . 10
2.5 Software system annotated with VCC2 . . . 10
2.6 Pseudocode of the update_student() function during modular verification . 11
2.7 The birthday() function with corrected annotation . . . 12

5.1 The max() function . . . 31
5.2 The max() function annotated in ACSL for Jessie . . . 32
5.3 The max() function with annotations and pragma . . . 33
5.4 Output of the max() function verification with Jessie, pragma SeparationPolicy is set to none . . . 34
5.5 The complete C programm with declaration, implementation, and invocation of the main() function. The invocation has some aliased parameters. 34
5.6 Complete C programm with max() function ported to VCC . . . 35
5.7 Output of VCC for the code in Listing 5.6 . . . 36
5.8 Inferred verification conditions in Why . . . 38
5.9 The C program with extended VCC annotations . . . 39
5.10 Generalized sample code for alias detection, pseudo-code . . . 41

A.1 Jessie code of the sample C program . . . 49
A.2 Why code of the sample C program (abridged) . . . 50
A.3 Source code for the DoParse() function of Jessifier . . . 57
A.4 Regular expressions used for code parsing . . . 60

Index

abstraction levels, 37
alias detection, 30, 46
aliasing, 2, 17, 25, 28, 31, 40
  complete, 25
  partial, 25
assertion, 7, 11, 19
assumption, 11, 19

bug-finding, 1, 4

call-by-value, 32, 40
coding discipline, 25
contracts, 6
  invariants, 8, 19
  post-condition, 7, 9, 19
  pre-condition, 7, 19, 32

debugging, 5

finite-state systems, 5
Frama-C, 15, 30
  Jessie, 15, 16, 21, 45
    optimizer, 45
  Why, 16, 45

hardware validation, 22

infinite-state systems, 5
intermediate language
  C, 13, 20
  proof language, 13, 20, 22

Jessifier, 41

max(), 31–34
  VCC, 35, 39
memory model, 24, 25, 33
  explicit specification in Jessie, 28
  Jessie, 21, 27
  ownership, 27
  VCC, 27
memory reinterpretation, 17, 24
memory safety, 1, 24
  language-based, 24

pointer, 25
  null pointer, 6
  typed pointer, 25
  valid pointer, 6
  void pointer, 25
pragma, 33
preprocessor macro, 7, 16, 18, 41
proof assistants, 20
proof obligations, *see* verification conditions 8

SAT solvers, 22
  MiniSAT, 22
semantic blur, 17
separation analysis, 2, 25, 28, 31, 33, 45, 46
separation logic, 2
side effects, 22, 31, 40
SMT solvers, 22, 28, 46
  Alter-Ergo, 16
  background theories, 22
  CVC3, 23
  Ergo, 23
  Fx7, 23
  Jessie support, 23
  preamble, 22
  separation logic prover, 28, 46
    jStar, 2, 28, 46
  Simplify, 22
    file format, 16
  SMT-LIB, 16, 46
  Yices, 23
  Z3, 16, 23
software specification, 1, 5, 6
  ACSL, 7, 15, 18
  inference, 20, 25, 38, 41, 46
  JML, 15
  strengthening, 39
  VCC, 7
  weak, 9, 11
software validation, 4, 5
  sound, 4

software verification, 2, 4, 5, 11, 46
  algorithmic verification, 5
  deductive verification, 5, 8, 15
  dynamic verification, 7
  modular verification, 9, 10
  push-button verification, 8
  software testing, 7
  static verification, 8, 13
  workflow, 13
strong typing, 24

tautology, 22, 45
type safety, 24

VCC, 15, 16, 30
  Boogie, 16
  VCC3, 15, 16, 18
verification conditions, 8, 45
  generation, 22

weakest pre-condition, 9, 20

Chapter 1
Introduction

> The required techniques of effective reasoning are pretty formal, but as long as programming is done by people that don't master them, the software crisis will remain with us and will be considered an incurable disease. And you know what incurable diseases do: they invite the quacks and charlatans in, who in this case take the form of Software Engineering gurus.
>
> Edsger W. Dijkstra, [Dijkstra 97]

This work addresses some aspects of the vast domain of software dependability analysis (often restricted to safety-critical software in avionics, medical software, and others [Baumann 10, Baumann 09b, Baumann 09a]), whose primary goal is to prove with higher degrees of confidence the ability of a software system to deliver an appropriate service [Moy 09]. Dependability can be further subdivided into several branches depending upon the purpose of the software under test and is generally characterized by the following features: reliability (robustness of the service), safety (as regards consequences for user), integrity (communication with other programs and the system), and security (flawlessness). In our work, we direct our attention to the memory safety aspect of software dependability, an important part of a software integrity check.

The modern technical progress leads to a massive spread of software use, even in systems in which one previously did not expect to see any software (embedded systems, microcontrollers, etc.). Especially in mission-critical systems where a software failure may result in enormous financial or human losses, this development poses additional requirements on the software quality. Formal verification of software systems comes here to the front as the only approach that can rigorously discover potential bugs and help eliminate them.

However, software validation is neither a magic wand nor a code review done by an oracle. If we want to verify our code, we first need to describe our expectation about what the code is supposed to do. Only then we might be able to detect possible bugs by comparing the expected behavior with the actual one. These expectations, turned into a set of conditions or obligations our code agrees to fulfill, are called specification. Writing the specification for a software system is the very first step in any verification process. The complexity of a specification might be orders of magnitude higher than the complexity of the system under test [Leinenbach 09]. Therefore, mistakes in specifications are usually more difficult to find compared to mistakes in the source code (see Section 2.5 for an example of a specification mistake).

One of the possible reasons why specification mistakes are so difficult to find is because the specification language is not an executable language, it rather contains the set of rules

or assertions that must be satisfied for the program to function correctly. In this respect a mistake is not only a possibly false statement about the code body. It is rather an assertion or assumption (see for the difference Section 2.5) that does not confine the possible set of cases that satisfy this constraint to legal cases only, e.g., allowing some non-valid (and possibly not so evident) cases to pass the specification conditions' check. Specifications containing that type of mistakes are designated as „weak specifications" [Moy 09] and can lead to failures during formal verification of complex software systems. In this work, we tackle the problem of aliases: variables with different names sharing one memory location. Changing the value of one variable inevitably leads to a value change in all aliased variables and can dramatically impede dataflow analysis of the system. Besides, aliases promote so-called side effects that can be puzzling not only for a user, but also for a developer and thus lead to hardly detectable bugs. Detection and elimination of aliases and side effects helps one to make code cleaner and less error-prone, and are important tasks of software verification.

Fortunately, mathematical models of computer memory [Cohen 09c, Hubert 08, Botincan 09] allow us to perform separation analysis and make sure that there is no aliasing possible in the execution context of the current software system under test. These findings can be further expressed in form of new contracts and added to the existing specification, thereby strengthening it (see Section 5.4). The improved specification allows us to approach the problem of alias detection more effectively.

Despite the fact that aliasing is a common problem in the analysis of C programs, certain verification toolkits might be optimized for other types of analysis. A good example is the VCC toolkit, which is designed with verification of concurrent C programs in mind [Cohen 10c]. This design consideration was extended to a so-called ownership model (see Section 4.3) [Jacobs 11, Cohen 09b]. These modifications allow for easier control over the access rights to every given memory location, but at the same time hamper alias detection.

The verification toolkit, however, can also benefit from the separation analysis results that reveals possible aliases and helps to add implicit constraints. Alias detection is then performed by another, foreign verification tool and its results are directly incorporated in the current specification. Thanking to the extensible architecture and clear separation of different system parts in VCC, it is possible to use a custom-built memory model during verification and thus extend the assortment of available analyses in VCC. Every memory model, however, needs to be implemented from scratch separately for every tool with almost no possibility to use an existing implementation (if any) and just adapt it to the interface of the system.

A custom implementation of a memory model was used by Matko Botincan with colleagues [Botincan 09] to implement support for separation logic verification in VCC. Separation logic is a novel, promising approach to program verification that allows one to specify and verify properties of the allocated memory both on the heap and stack by reasoning about different memory footprints, which also includes alias detection. The program specification was automatically split into a general VCC specification and a specification written in separation logic. The latter was verified using a special separation logic prover named jStar, which evaluated separation logic expressions and checked them for validity.

Unfortunately, the project was discontinued and the toolset is no more compatible with the current version of VCC and jStar [personal communication, 2011]. That means that currently there is no verification tool in the VCC toolset that would allow alias detection.

The primary goal of this work was to employ an existing implementation of the separation analysis to strengthen specifications by adding new, automatically derived constraints.

We used Jessie, a deductive verification plug-in for Frama-C that supports separation analysis to perform alias detection. The ultimate goal is to automate the inference of these constrains and to integrate them directly as annotations into the VCC specifications to arm the latter with alias detection mechanism. We refer to this strengthening of the original VCC specification as „alias contract inferring."

During the work on this thesis we used the following versions of the verification tools: Microsoft Research VCC Version 2.1.40226.0, Boogie Version 2.1.21217.0 and Z3 Version 2.0.41203.1. For separation analysis the latest version of the Frama-C (Carbon-20110201) and Jessie/Why plug-in (Version 2.29) were used.

Due to the tight time frame of the current work we do not implement a plug-in for VCC and provide just a basic automation for this type of annotation inferring. It is especially difficult to reconcile these two verification toolkits because they are optimized to run on different operating systems. We set up different virtual environments for every toolkit and use the benefits of both tools for quick and robust data verification.

This thesis consists of 6 chapters including the current Introduction (Chapter 1).

In Chapter 2, we describe the basic principles that form the mainstay of software verification, discuss the difference and frequent misuse of the terms „verification" and „validation", discuss possible approaches to software verification, peek into code annotation languages and possible annotation kinds, and learn the fundamentals of deductive verification. The chapter is closed by an example of modular verification that illustrates how a subtle mistake in a specification can easily have dramatic consequences. It is the modularity in verification that accounts for the majority of failures and that one has to master first to write verifiable code annotations.

In Chapter 3, we pay attention to the verification process in general and to both verification toolkits, VCC and Jessie, in particular. We try to find the similarity and differences in how they function and what the benefits are of either system. We learn the chain of tools that are applied subsequently to the source code to perform the various steps of verification analysis and will see how to interpret their results.

The next chapter, Chapter 4, deals with different approaches to memory modeling as a key prerequisite for separation analysis and alias detection. These two terms are also thoroughly explained here.

In Chapter 5, we present our own contribution to the topic of the thesis. We first derive a short illustrative example of the `max()` function that can be successfully analyzed using Jessie but not using VCC. We analyze the intermediate results from both toolkits to find the additional constraints Jessie adds to the provided specification of the code to prove it valid, extract them from the code in its intermediate proof language, and integrate these assertions into the original specification of the C program. This allows us to use the benefits of alias detection from Jessie without the need to re-implement separation analysis in VCC.

In the last chapter, we discuss the results presented in Chapter 5, evaluate the advantages and limitations of this solution, and review work on related topics.

The complete source code of several important routines is provided in Appendix A at the end of the document.

Chapter 2
Software validation and verification

Software validation[1] is a complex formal act of proving correctness of a hardware or software system [Moskal 09]. The interpretation of „correctness" usually takes the two following aspects into account:

1. The system must be free of runtime errors, that is, it should not crash or freeze during normal operation (as judged by the user) and should not contain any illegal operations (division by zero, or memory access outside of the allocated area)
2. It must adhere to its specification, defined implicitly (assertions in the program code) or explicitly (contracts), inline or externally (rule-based specification).

The correctness can be proved only under certain assumptions made in respect to the compiler and runtime environment as regards the semantics of the language. One very common assumption about a language is the memory model used to simulate (and to verify) memory access in the program; we discuss several approaches to memory modeling in Chapter 4. Other assumptions refer to the type system (and especially to type safety) of the language, possible integer over- and underruns, etc.

If these assumptions are guaranteed by both compiler and runtime environment, i.e., they hold for every possible execution of the program code, then the validation is said to be *sound*. Otherwise, if the assumptions can be enforced only for certain „common" execution scenarios, the verification approach is called *bug-finding* [Moskal 09].

2.1 Software validation vs. software verification

Verification and validation are two terms that are often used concomitantly or interchangeably in software engineering research. This combined used of two paraphonic terms is so wide-spread that they are often used as an abbreviation V&V [Pierce 96].

However, it is important to understand that these two terms are not identical – they refer to two distinct but complementary activities. Software verification provides objective evidence that the outcome of a particular phase of the software development life cycle meets all the requirements for that phase provided in the specification.

Verification is performed by checking for consistency, completeness, and correctness of the software and its supporting documentation [Jetley 09]. It can also include the following

[1] See also section 2.1 for the definition of validation and verification.

types of activities: all types of testing activities (different types of tests, both human-written and automatically generated), symbolic execution of the program code, calculation of software metrics, profiling software performance and memory consumptions. Software verification is entirely based upon the given specification, takes its correctness for granted, and attempts to answer the question: „Does the software at the given stage of maturity comply with every condition required by the specification?" or colloquially speaking „are we doing things right?"

Validation, on the other hand, is the confirmation by examination and provision of objective evidence that software specifications conform to user needs and intended uses, and that the particular requirements implemented through software can be consistently fulfilled. Here we turn our attention to the specification and can apply to it the same activities we previously used to verify our code, trying to make sure that the specification adequately represents the user requirements, both implicit and explicit, expressed in any human-readable form. This necessity for software validation arises from the fact that software specification, as any machine-processed code, is bug-prone and requires debugging and correction. Compared to implementation bugs, bugs in specifications are usually due to the implicit nature of some requirements that are not easy to discover and express in an explicit form – the only form verification tools can understand and use for verification.

The process of software checking is constituted by alternating validation and verification to keep the code and specification in balance. This alternation will become more evident while reading this thesis, where both specification and implementation get refined. In this respect we will use the terms „validation" and „verification" interchangeably and try not to confuse the reader with the peculiarities of the both.

2.2 Approaches to software validation

Software validation employs several principally different approaches to prove certain software systems correct (or not). The decision point for selecting an appropriate methodology here is the complexity of the system or, more precisely, the finiteness of its state. For finite-state systems, which are unfortunately quite rare in software validation compared to hardware validation [Moskal 09], one can employ both deductive and algorithmic methods to check the system for correctness (see Fig. 2.1). These methods include the approaches to fully enumerate all possible system states or symbolically execute the program code to check for possibly unexpected results.

Infinite-state systems, which are the common case in software development, are amenable only to deductive methods, which require much more effort and competence from the developer to perform validation. Besides the source code to verify one has to provide the expectations this code has to meet, formulated in the way that can be proved automatically or semi-automatically. These expectations, written in form of mathematical formulas, are called specifications.

Verification of a software system is then an attempt to make sure that the system under test meets every single requirement (expressed as a condition) from the specification provided. It is important to understand here that a failure during this attempt should not necessarily be blamed upon the source code. Specifications are at least as error-prone as the implementation itself (as you will see on page 12), and can contribute to verification mistakes. Successful verification is not a *panacea,* too, because it might indicate that the

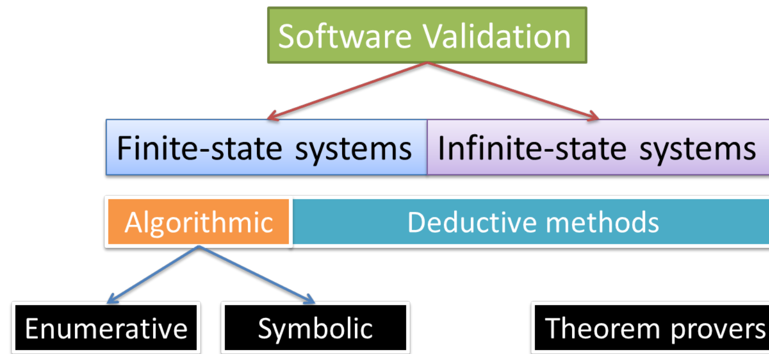

Figure 2.1: Approaches to software verification

specification is not strong enough to reveal bugs or does not adequately implement the real-world constraints or requirements.

As one can see, software validation is somewhat tricky and requires not only competence and experience, but also attention and patience. It is therefore not surprising that special positions for specification engineers exist in large developer teams.

2.3 Specifications and contracts

As stated in the previous section, we first need to decorate our code with statements describing the expected behavior of the program if we want to perform any verification of this code. Let us take a simple, naive implementation of an `abs()` function that returns the absolute (e.g., positive) value of the given signed integer as shown in Listing 2.1.

```
1 int abs(int *a) {
2   if (*a < 0) return -(*a);
3   else return *a;
4 }
```

Listing 2.1: Simple implementation of an abs() function in C

This program reads an integer value from the memory location specified by `a` and returns its absolute value; the original value is not modified.

Now we need to write a specification for this rather straightforward piece of code. To achieve this goal we need to identify the assumptions we make in our source code and write those down explicitly. It might seem rather difficult at first glance to see what kind of assumptions we made. A valid approach to derive a specification is then to seek for the test cases that could potentially crash our code. Passing „null" as the input value will lead to a crash, as well as providing a pointer value referencing to outside of the memory allocated to the application. The condition for a pointer to address a legal memory location is common in software verification and is usually expressed by an internal Boolean function $valid(x)$ that ensures the „validity" for the given pointer `x`. Now we can write our first specification condition as `assert(valid(a));` as the very first line of the code, between line 1 and line 2 in Listing 2.1.

Similarly, we expect that every value returned by the function is greater or equal zero. We can then add a second „`assert()`" before every return. In this „`assert()`" we need to

access the returning value of the function, which can be done by introducing a temporal local variable that will contain the returning value. The assertion about the return value is placed before every exit point of the function. You can see the modified code in Listing 2.2.

```
1 int abs(int *a) {
2   assert(valid(*a));// <- 1st assertion
3   int c;
4   if (*a < 0) c = -(*a);
5   else c= *a;
6   assert(c >= 0); // <- 2nd assertion
7   return c;
8 }
```

Listing 2.2: Implementation of the abs() function in C decorated with assert() statements

This code remains valid C code[2], for `assert()` is a preprocessor macro from the standard C library (defined in `assert.h`) that checks the condition at runtime. In case it does not hold (e.g., its evaluation returns zero) the error message is written to `stderr` and program execution is aborted. The use of `assert()` statements in C code may be seen as a good practice, because these assertions introduce runtime constraints and make possible the dynamic verification of source code. It is important to understand that dynamic verification does not fall under the set of formal verification methods but belongs to software testing.

If we now critically go over our two assertions, we see that the first assertion indicates the condition that must hold before the execution of the code fragment (in our case, before function invocation) and that the second assertion illustrates the condition that will hold after correct execution of the code (fighting against the principle „crap in, crap out”). The first condition is then called „pre-condition” (because its check *precedes* the actual execution of the code) and the second one is called „post-condition” (for it holds after the code has been executed).

Due to their fixed location in the source code (just before and after the function body) they can be easily combined and moved towards the very beginning of the code block invocation, either just before (ACSL specification syntax, used in Jessie) or just after (VCC syntax) the function signature, where they are called „annotations” or „code contracts.” To enable backward compatibility to the native C code and to keep the source code compilable by a conventional C compiler, the annotations are either written as C-style comments (ACSL), or commented out using a preprocessor macro that is redefined by the verifier (VCC). With the purpose to distinguish between the invocations of the source code and internal functions of the verifier, the latter ones are usually prepended with backslash (ACSL) or underscore. The complete listing of the ACSL-annotated `abs()` function you can see in Listing 2.3.

[2] If you want to try this code and get the compiler error that `valid()` is not defined, use the following macro definition in your header file: #define valid(condition) (condition != null)

```
/*@ requires \valid(a);
  @ ensures \result >= 0;
  @*/
int abs(int *a) {
  if (*a < 0) return -(*a);
  else return *a;
}
```

Listing 2.3: The abs() function with ACSL annotations

Code annotations allow the static verification of the program to use formal methods. That means that we can verify the correctness of the program given its specification, without any need to run the program. The outcome of this analysis is then more general (and thus potentially an overapproximation), detecting not only *possible*, but also *potential* bugs in the code that might never be triggered. To rephrase the famous quote by Dijkstra, this analysis strives for the absence of the bugs, not for their presence[3].

Besides the conditions that hold before and after the code execution one may also want some conditions to hold during the complete execution of the program code: for example, time and age can never turn negative, irrespective of the operations performed. Those contracts are known as invariants and can be easily outsourced into separate files which contain only specifications and no source code, and are usually located in the same folder where the source code files reside. Even further separations of those contracts into the set of rules the software should fulfill is called rule-based specification. These sets of rules are very common in software quality assurance, for they enable easy push-button verification of the source code. This approach is used, for example, in the static verification of Windows drivers [Henzinger 03, Ball 04].

As soon as the code is annotated we can start with its verification using formal methods of mathematics, especially first-order logic.

2.4 Deductive verification

The most general idea behind deductive verification is to convert the proof for software correctness into the proof of logical formulas [Moy 09]. These formulas, called Verification Conditions (VCs) or Proof Obligations (POs) are then automatically „solved" (i.e., proved correct or wrong) by theorem provers.

The origin of verification conditions are the annotations, which are resolved in the source code and converted (if needed) into logical formulas. Importantly, there is no need to have a ready implementation to perform deductive verification: during verification we make sure that the source code does not violate any of the conditions provided in the specification, and empty function or method bodies are the perfect candidates for verification, for they can violate nothing. This makes possible a top-down approach to the design of software systems, when the specification and the outline of the source code (declarations etc.) are written first and get validated, and the actual implementation just follows. This approach does not protect against possible bugs in the implementation, but allows one to check the system design and discover architecture and design bugs already during the design phase of software development.

[3] „Program testing can be used to show the presence of bugs, but never to show their absence!" [Dijkstra 72]

One of the main principles used to convert the contracts into verification conditions is the one suggested by Dijkstra's „weakest pre-condition" approach [Wand 77]. We can mathematically express a code fragment annotated with pre- and post-conditions in the following form:

$$\{precondition\}\ Expression\ \{postcondition\} \tag{2.1}$$

In this equation our complete code is abstracted as an expression; the equation can be seen as „correct" if the post-condition holds whenever the pre-condition holds. The only evaluation of the expression that might lead to the change of condition is the assignment of a variable that existed before the expression and is contained both in pre- and post-condition. We can rewrite our equation then as an assignment expression,

$$\{precondition\}\ x := F(\xi);\ \{postcondition\} \tag{2.2}$$

where ξ is the program state and $F(\xi)$, the function of this state; „:=" denotes the assignment of the variable with a new value. In the next step, we express the post-condition from (2.2) to hold as the set of (in)equations (predicates) $P(x)$ containing the variable x:

$$\{precondition\}\ x := F(\xi);\ \{P(x)\} \tag{2.3}$$

In this situation we can perform a backward transformation of the assignment from (2.3) with the purpose to find the weakest (read: the most general) pre-condition, thereunder the assignment of x will not lead to any violation of the post-condition:

$$\{P(F(\xi))\}\ x := F(\xi);\ \{P(x)\} \tag{2.4}$$

As soon as the weakest pre-condition is derived, it can be checked against the given pre-condition and corrected if necessary. The weakest pre-condition approach depends largely upon the high-level abstraction of the source code expressed as the function $F(\xi)$. To enable robust and performant solving of the pre-condition, $F(\xi)$ is required to be a quantifier-free first-order logic formula, which forces modern verification systems to employ sophisticated methods of mathematical logic like E-matching to reach this level of abstraction.

Performance issues also account for the so-called „modular verification" of the program code, discussed in the next section.

2.5 Modular verification

Modular verification is a commonly used technique in software verification when the function body is evaluated only once and the contracts resulting from the weakest pre-condition algorithm are then stored and used to verify other fragments of code instead of the source code. This saves the time for re-evaluating of already proved code and boosts the overall verification performance, but can have disastrous consequences if there are errors in the specification, especially if the specification is not strong enough for the given code.

We try to illustrate this danger by giving an example[4]. For the sake of simplicity we use signed integer as the only data type, disregarding all composite and referenced data types.

[4] All examples in this chapter are written using VCC2 annotation syntax. This was intentionally done to attract the reader's attention to VCC's verification model, for in that case the potential danger of a weak specification is mitigated by the annotations inferred using the region-based memory model in Jessie.

The code in Listing 2.4 illustrates the function `birthday()`, which might be a database stored procedure or data layer maintenance function that performs an update of the person's age by returning a new, incremented age given the current age.

```
int birthday(int oldage)
   requires (0 <= oldage && oldage <= 99)
   ensures (1 <= result && result <= 100)
{
  return oldage + 1;
}
```

Listing 2.4: The birthday() function annotated with VCC2

We can apply the techniques explained in the previous section and easily derive the pre-condition from the post-condition by simply decrementing the upper and lower bounds for the function return value.

This function seems to be correct as regards both its source and specification, and if we perform verification, both tools (VCC and Jessie) return positive verification results.

Now if we need to verify a more complex system containing this function we can come up with the following sample code shown in Listing 2.5.

```
struct Student {
  // ...
  int Age;
  datetime Birthday;
  // ...
};

int birthday(int oldage)
   requires (0 <= oldage && oldage <= 99)
   ensures (1 <= result && result <= 100)
{
  return oldage + 1;
}

void update_student (Student stud)
        requires (16 <= stud.Age && stud.Age <= 65)
        ensures (16 <= stud.Age && stud.Age <= 66)
{
  // ...
  if (stud.Birthday == today()) {
        stud.Age = birthday(stud.Age);
  }
  // ...
}
```

Listing 2.5: Software system annotated with VCC2

Here, `birthday()` represents the function called from the `update_student()` routine to perform all necessary actions and get the updated age information.

Due to the modular nature of software verification in the system under test, `birthday()` has no external dependencies, so it will be verified first, and verification succeeds. In the

second step, the `update_student()` function will undergo the verification. Because the latter contains an invocation of an external function that has been already verified, the invocation is then substituted by inlining the function contract, leading to the (pseudo)code shown in Listing 2.6.

```
1  void update_student(Student stud)
2  {
3          assume(16 <= stud.Age && stud.Age <= 65);
4          // ...
5          if (*)
6          {
7                  assert(0 <= stud.Age && stud.Age <=99);
8
9                  // stud.Age = birthday(stud.Age);
10
11                 assume(1 <= stud.Age && stud.Age <=100);
12         }
13         // ...
14         assert(16 <= stud.Age && stud.Age <=66);
15 }
```

Listing 2.6: Pseudocode of the update_student() function during modular verification

Here we meet a new verification directive `assume()`. Similarly to its counterpart `assert()`, this directive enriches our knowledge about the system under test by adding some new facts or confining the existing ones. But compared to `assert()` we do not prove the validity of the new facts and rather take them „for granted." This means, that if we need to narrow down the interval of possible values for any program variable we can easily do so with `assume()`, whereas the same facts supplied using `assert()` would trigger an exception. Thus, `assume()` allows both widening and narrowing redefinitions of the known fact about the system under test, whereas `assert()` allows only widening redefinitions.

Let us analyze the code in Listing 2.6 step by step. In line 3, we first define the possible interval for the age variable. In the original code we have a conditional branching in line 5 that depends upon the current date. Since software verification (compared to testing) does not stick to a particular execution of the code and rather tries to generalize the correctness for every possible case, we can eliminate every condition here. The asterisk that we used to substitute the original condition is a universal condition that can be evaluated to both true and false, which forces the verifier to take all possible branches in the source code.

The assertion in line 7 is just a widening redefinition and does not violate anything. Following the redefinition of the age interval in line 11 we finally bump into the assertion in line 14, where we need to narrow down our broad definition to a somewhat more compact interval and our verification fails here.

The reason for the failed verification is the specification of the `birthday()` function, which was not strong enough. This „weakness" does not manifest itself during the verification of the function. It is only revealed when the outcome of function verification, consequently not strong enough too, is used to verify another function where this weakness cannot be tolerated, like `update_student()` in our example. Indeed, the post-condition of the function allows the return value to lie within a very broad interval irrespective of the input value. Even though the boundary conditions are still met, e.g., for corner cases when the input value represents the minimal and the maximal acceptable values, these boundaries

are also propagated to the return values; the current specification allows for example to return 5 (return value) in response to 20 (`oldage`). How counter-intuitive and unreal it might look, this outcome meets all the requirements denoted by the specification, pointing out to a potential error in the latter.

The correct specification that makes use of the supplied parameters and does not needlessly widen the interval of acceptable return values is shown in Listing 2.7.

```
1 int birthday(int oldage)
2    requires (0 <= oldage && oldage <= 99)
3    ensures (result == oldage + 1)
4 {
5   return oldage + 1;
6 }
```

Listing 2.7: The birthday() function with corrected annotation

Taking this rather trivial example, one can easily see for oneself that specifications are even more error-prone than the source code and finding bugs in the specification might be even much more challenging than finding bugs in the source code.

Chapter 3
Verification tools and tool chains

The general architecture of all software verification tools is very similar. In the very first step, the source code is translated to a side-effects free intermediate language, like the C Intermediate Language (CIL[1]) in case of Frama-C. Then, the code is converted into a so-called intermediate proof language, which incorporates all specification conditions from the original code encoded and expressed as formulas so that all parts of the system are abstracted as global or local variables (stack, heap, registers). Verification Condition Generators (Why in Frama-C and Boogie in VCC) then usually generate first-order logic formulas out of these intermediate language abstractions, which can further be automatically proved (or disproved) by a theorem prover, or semi-automatically proved and analyzed using proof-assistant systems.

Different verification systems, however, use slightly different approaches to verification. This divergence is especially evident if we compare the memory models used (see Chapter 4); other differences might be the SMT provers used to prove the obligations and the types of analysis that verification tools are able to perform. In this chapter, we will analyze two different verification tools trying to discover their distinctive traits.

3.1 Software verification workflow

As it has been already discussed in the previous chapter, software verification and the correction of a specification are not a trivial process and often require the repeated execution of the verification routine with specification adjustment after every execution. In this respect, the overall strategy of the verification seems to be similar to „trial and error"-approaches where the specification engineer refines either the specification (preferably) or the source code (if a bug was detected and confirmed) until both finally become coherent and the software verification of the complete system finally passes with no error or warning.

The typical workflow for software verification is depicted in Figure 3.1. Previously annotated code (see Section 3.2.2 for more details) is fed into the verification tool with the goal to perform static software verification using code annotations. As a result of the verification, three outcomes are possible: software is proved to be bug-free (1), the condition could not be verified and an error message was generated (2), or the verification calculation timed

[1]Do not confuse this language with the Common Intermediate Language (CIL), formerly known as MSIL (MicroSoft Intermediate Language), an object-oriented, virtual-machine based assembly language.

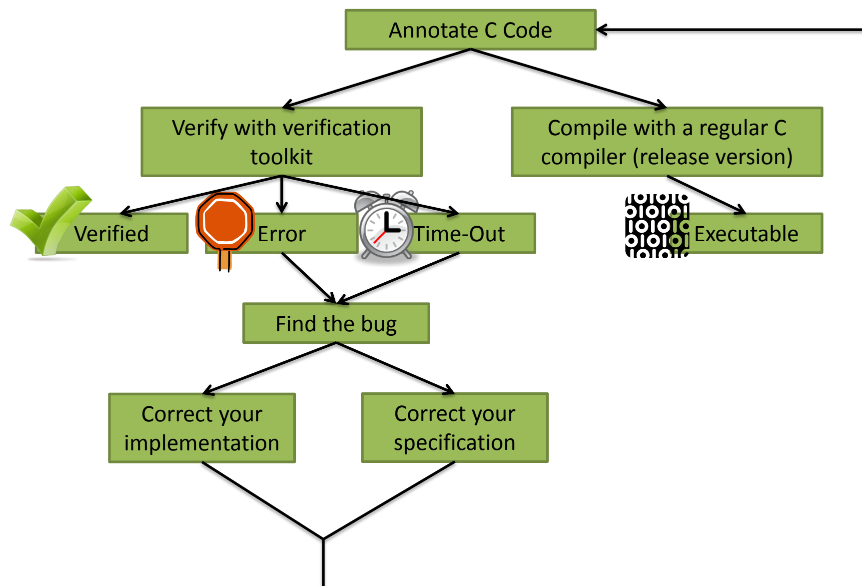

Figure 3.1: Typical workflow in deductive software verification illustrated by tools around VCC (adapted from [Cohen 09b])

out (3). Any other outcome, like „unknown" (? in Jessie) and „invalid" (*) just cover possible communication problems between the VC generator and the SMT-prover

Let us briefly discuss these three main possible outcomes.

3.1.1 Outcome: Verification was successful

A successful verification is usually a long-awaited result one reaches after numerous iterations of the refine-and-re-verify cycle. In this case, this positive result usually indicates that the software under test is error-free and suits the derived specification.

There is still a chance that the specification might be not as strong as necessary to detect all possible problems, but multiple refinement steps and a well-documented detailed specification speak against that possibility.

Due to its modularity, modern verification is neither associative nor commutative, meaning that the successful verification of every system part does not imply that the overall system verification will be successful as well and that any subtle change in the relations between parts („a calls b" vs. „b calls a") drastically influences the final verifiability of the system.

3.1.2 Outcome: Verification fails

Failure during verification is the most common outcome during the very first steps. A failure indicates that source code or specification contain errors and that the code does not match the conditions from the specification.

In many cases, there is a way to retrospectively analyze the trace that led to the failure and come up with a counterexample that crashes the code (illegal arithmetic operation, memory access violation, etc.). If the counterexample can not be easily derived it may imply a bug in the specification. Depending upon the reason of the failure one of the two needs to be corrected and the verification check is repeated until one obtains a positive result.

3.1.3 Outcome: Verification timed out

Timeout is inconclusive and indicates the inability of the SMT-prover to find an acceptable solution (or disproof) for the proof obligation. This might be due to an overcomplicated expression that requires too much computational time and memory to be analyzed, or due to the problems with background theories (see page 22) of the given SMT-prover. Possible solutions are revising the annotation that caused the timeout, trying to simplify the specification by dividing it into several clauses with a simpler structure, or moving it to another place in code. In case of problems with background theories, one might suggest using another SMT-prover; both VCC and Jessie support multiple output formats and can be combined with different provers (see the list in Section 3.2.6).

Often, timeouts during verification imply problems with the theorems and axioms provided by the intermediate proof language, the SMT-solver, or both, and might generally indicate that the given verification tool is not well-suited for verifying such types of conditions. On the other hand, developers behind the verification tools tend to profile their axioms and theorems and revise the preamble theorem set in every new version of their tool; for example VCC3 will contain only half as many axioms and theorems compared to VCC2 [Cohen 10c].

3.2 Software Verification Tool Chain

In this work we use two different software verification tools. One tool is called Frama-C, developed by two French public institutions: CEA-LIST (Software Reliability Laboratory) and INRIA-Saclay (ProVal team, common with LRI-CNRS and Université Paris-Sud) [Marché 10]. Frama-C is an open source platform for source code analysis of software written in C, with a pluggable, easily extensible architecture. The platform itself provides the initial parsing of C code and its conversion to side-effect free, standard-compliant C. Conversion to CIL keeps the original semantics of the program as well as all annotations that can be made in the original source code using the ACSL annotation language.

Frama-C provides just the initial pre-processing of C code and does not implement any kind of code analysis, leaving all types of analysis to its numerous plug-ins. Worth to mention are the plug-ins for value analysis, deductive verification, impact analysis, scope, data-flow, and variable occurrence browsing. Jessie is one of the plug-ins for deductive verification (see Section 2.4) developed by INRIA-Saclay that provides memory safety checking, integer overflow, and termination analysis. It performs functional verification of the program annotated with ACSL and implements separation analysis. Experimental features of the plug-in include the inference of annotations, both pre- and post-conditions, using several variants of the weakest pre-condition technique (see page 8) [Marché 10, Moy 09].

Jessie is implemented as a plug-in for Frama-C and enables analysis of C code with ACSL annotations. At the same time, it is suitable to analyze Java code with JML annotations

with the help of another preprocessing tool, Krakatoa [Moy 09].

Jessie as plug-in contains two translators: one translates the CIL code with annotations, the output of the Frama-C framework, into valid Jessie code. This step is needed to enable compatibility between Java/JML/Krakatoa and C/ACSL/Frama-C verification paths and to reach an adequate abstraction for annotations provided by two different annotation languages (see Section 3.2.4).

The source code in Jessie is translated to Why using the Jessie2Why compiler. This compiler accounts for the generation of proof obligations, which are then proved using a SMT-prover of choice (for fully automated verification) or using an interactive verification tool. The packaged Frama-C installation also contains the Alter-Ergo SMT-solver, which is used as a default automated theorem prover (ATP) in Jessie/Why. However, Jessie/Why provides also a built-in support for five other SMT solvers, including the Z3 solver from Microsoft Research.

Besides Frama-C there is a similar tool chain from Microsoft Research, called Verified C Compiler (VCC) [Cohen 09a, Cohen 09b, Cohen 09c, Cohen 10a, Cohen 10b, Cohen 10c, Maus 08, Moskal 09]. Based upon the Common Compiler Infrastructure (CCI) from Microsoft Research, VCC combines a general-purpose C compiler with a verification platform. VCC was originally designed to extend the C language with contracts, similarly to those languages that have a native support for contracts: Eiffel, Cω, and Spec#. There, contracts are then placed directly in the function signatures. This annotation format leads to incompatibility between ANSI C and the C code annotated for VCC, the latter is then the only compiler that still can compile this code. To circumvent this problem in the yet-to-be-released major version of VCC, termed VCC3, the annotations are wrapped into the brackets with preceding underscore („_(annotation)"), which allows one to define a preprocessor macro that automatically comments out the lines with annotations, rendering the annotated C code compilable by a conventional C compiler.

VCC uses Boogie, another language developed by Microsoft Research, as an intermediate proof language that generates verification conditions. This language is used by several other Microsoft Research projects besides VCC including Spec#, HAVOC, SymDiff, and Verve. The default output of Boogie is set to the Simplify file format (.sx) (see Section 3.2.6 for details) that is then used as input file for the Z3 SMT solver from Microsoft Research delivered together with VCC. Newer versions of Boogie and VCC will completely switch from Simplify to SMT-LIB for their internal file format for verification conditions, to enable the use of the broad range of SMT solvers understanding SMT-LIB format. As of today, the SMT-LIB output of Boogie is still experimental and the generated file cannot be processed using Z3 or any other solver.

In the following tool chain

$$(input)\ C\ code \Rightarrow \frac{Frama\,C \Rightarrow \frac{Jessie}{Why}}{VCC \Rightarrow Boogie} \Rightarrow SMT\ Solver\ (output)$$

the abstraction level rises together with the size of the code, which increases by approximately an order of magnitude in every subsequent abstraction step. The „puffing" of the code is due partially to abstractions from the original code, partially to preambles (axioms, theorems, definitions etc.), which are language-specific and inserted at the very beginning of the code file by the translator at every stage.

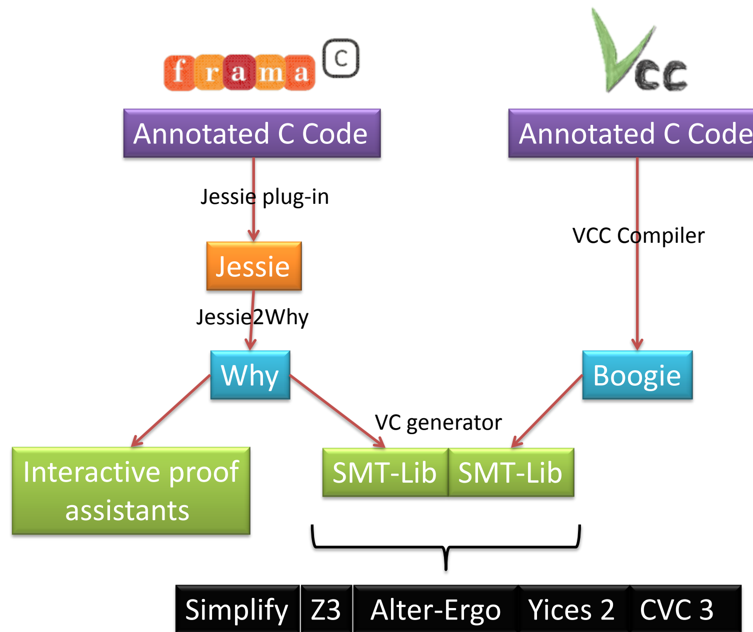

Figure 3.2: Comparison of the software verification tool chains of Jessie and VCC

3.2.1 Original C Code

The main problem of C code verification arises from the fact that the C language was originally designed by Dennis M. Ritchie at Bell Labs with two different and somewhat contradictory goals in mind [Moy 09]. On the one hand, C was regarded to serve as mere one-level abstraction above Assembler to easily write portable (e.g., runnable in different address spaces and on CPUs with different instruction sets available) programs using the techniques common for low-level languages (direct memory access and manipulation, pointer arithmetic, self-modifying code, dynamic code generation, etc.). On the other hand, C provides the possibility to use higher levels of abstraction (variables, functions, structures) and contains the elements typical for modern high-level general purpose programming languages.

This second trait of C allows one to write C programs that are type safe (without memory reinterpretation), which is common in modern strongly typed languages like Java and C#. These programs can be analyzed with less effort, for several types of aliasing (see Section 4.2) are not possible here by definition. Unfortunately, there is no way to say *a priori* whether C programs are written using the first or the second approach. Thus, a general verification tool should be potentially capable of analyzing any given C program, e.g., it should be equipped with tools to analyze all possible language phenomena.

Besides the ambiguity of its design, C also demonstrates the lack of precise language semantics that is often referenced to as „semantic blur" [Moy 09]. Moreover, despite considerable work spent on standardization of the language, most compilers provide so-called „compiler language extensions" that account for co-existence of several C dialects (GNU C, Watcom C, Microsoft Visual C, etc.).

These two properties of C (design ambiguity and semantic blur) make it almost impossible to apply verification techniques directly to the source code, but require certain preprocessing,

which results in normalized, side-effects free representation of the C code. There are several those languages, the best knowns are GENERIC and GIMPLE (both used by GCC as intermediate C languages during compilation), CIL (C intermediate language), and SIMPLE. Frama-C converts the source code into CIL, the C Intermediate Language, whereas VCC uses the internal format of CCI, the Common Compiler Interface from Microsoft Research.

3.2.2 Annotating code

As already mentioned, to be able to perform any type of verification of a software system, we need to provide a specification for the system we are going to verify. There are many ways to define specification for a software system, from verbal description of the system requirements in a natural language to strict mathematical models. A common approach here is to decorate the source code with conditions, written as comments. These conditions serve as a reference point we can perform our analysis against. For custom-built software systems written in C the specification is usually written inline, along with the compilable and executable code, and is marked as commentary either directly (ACSL [Burghardt 11]) or by using a preprocessor macro (VCC3 [Dahlweid 09]). This process is often called „code annotating." These short annotations for every program method written, using a special annotation language, contain (or better say consist of) the specification clauses. The latter is just a machine representation of the first-order logic with internal functions and directives to access program global variables, allocated memory, variable state before and after invocation etc. These properties and the operators[2] of the first-order logic allow us to describe the expected program behavior and to exclude all invalid results or intermediate states.

Code annotations consist of three main contract conditions and might also contain some functions or macros defined by the respective verification tool. In Table 3.1, we provide a short overview[3] of possible contracts.

Contracts can use extended Boolean logic to formulate the conditions: in addition to the basic logical operators (AND, OR, NOT, XOR, etc.) they may also contain arithmetical operations, functions, predicates, quantifiers, and some functions pertaining to verification, like `\old()`, which returns the value for the variable supplied before the invocation of the current function (see Table 3.2).

The set of available functions and operators allows one to derive rather sophisticated code annotations and helps to formulate the system requirements tightly enough to exclude any possible errors and to detect bugs.

Even though traditionally annotations are supplied for functions, it is possible to write annotations almost at every site in code. This option especially concerns local annotations like `assert()` or `assume()` and annotations that are used to verify loops (loop variants and loop invariants).

Annotated code presents the mixture of two different languages: the underlying imperative or object-oriented „executable" language and the meta-language of annotations to be analyzed, not executed. In order to perform software verification we need to merge both into a consistent representation of the program model we would like to reach (based upon

[2] By „operators", we mean relations (predicates, connectives), functions and quantifiers.

[3] For illustration purposes we provide a merged overview of the keywords used in annotations to describe certain conditions and do not stick to a particular syntax of VCC or ACSL.

Condition type	Typical keywords	Description
Pre-condition	requires	Defines the condition that should be valid before invocation.
Post-condition	ensures	Defines the condition that holds after the invocation iff the pre-condition holds.
Invariant	(global, type, loop) invariant, maintains	Defines the condition that holds globally, for a given datatype or a given loop.
Local condition	assert, assume	Defines the condition that must be valid at a certain program execution point.
Memory access	assigns, writes, reads	Explicitly denotes the type of memory access to certain variables in the function.
Termination	terminates	Provides the pre-conditions for the function to guarantee termination (used to relax pre-conditions).

Table 3.1: Types of contract annotations

Function type	Typical symbols and keywords	Description
Logical	&&, ||, ^^, ~, ==>, <==>	Boolean operators (including IMP and EQU)
Arithmetic	+,-,*,/	Basic arithmetic operators
Comparison	<, <=, >, >=, !=, ==	Comparison between integer, logical and real datatypes
Quantifiers	\forall(), \any(), \exists(), \lambda()	Quantifiers
Internal functions	\valid(), \old(), \result, \nothing, \null	Verification-specific internal functions and values
User-defined functions	predicate	User-defined functions that can be re-used in annotations

Table 3.2: Functions and operators that can be used in contract annotations

our specification) and the current model (based upon the implementation). This merge is only possible at a higher abstraction level, called „intermediate proof language."

3.2.3 Annotation inference

As already discussed in Sections 2.3 and 2.5, writing strong specification can become a tricky task. Even slight uncertainties in the specification may, on the one hand, prevent correct programs from being verified and on the other hand lead to really dramatic consequences if some erroneous cases are not entirely excluded by the specification. This explains why any sort of automation in deriving specification is more than welcome. Inference becomes of paramount importance during semi-automated verification with the use of proof assistants like Coq or HOL [Böhme 08, Böhme 10].

Whereas VCC provides no support for annotation inference, Jessie implements several types of inference mechanisms, depending upon the type of annotation one would like to infer [Marché 10].

The most common method to infer annotation is to use Dijkstra's weakest pre-condition technique (described on page 8) to propagate constraints. Using this method we can derive the weakest possible pre-condition for any given post-conditions. If the annotation is not as strong as the pre-condition derived, it can be strengthened by adding some statements from the derived pre-conditions. Conversely, the propagation of the pre-condition through the function code can help to derive the minimal post-condition that must hold or to strengthen its annotation.

Depending upon the annotation language, code annotations can contain information about variables that are modified (using the directives `writes`, `modifies`, `changes`, `accesses`) and that remain untouched (`maintains`, `reads`, `invariant`) in the function body. This knowledge, augmented with a symbolic execution of the function code, can be very valuable in software verification because it allows us to partially derive missing specifications and complete the weak ones.

Jessie implements the weakest pre-condition approach together with quantifier elimination and some heuristics (see Chapter 4 in [Marché 10]) to improve the precision of the inference.

3.2.4 Conversion from annotated C to intermediate proof languages

We illustrate the process of verification condition generations using only Jessie. In its counterpart, VCC, the generation takes place similarly and we do not pay additional attention to this tool.

To keep track of the transformations that take place during the conversion from the annotated and preprocessed C Intermediate Language to Jessie (see Figure 3.2 on page 17) we take the `max()` function from Chapter 5, the fully annotated code of which you will find as Listing 5.3 on page 33. The full source code of the Jessie program is too long to be provided here. You can find it in Appendix A.2 on page 49.

First of all, let us just take a swift look at the complete source code of the Jessie program. As one immediately can see it is prepended with a short preamble containing axioms and type definitions, followed by an abstracted representation of the original C code. It is quite easy to associate the lines of the original C code with their translation to Jessie, for both language share similar constructs (`type`, `if`, `else`, `return`). Other keywords in

```
/*@ requires \valid(i) && \valid(j);
  @ requires r ==\null || \valid(r);
  @ ensures *i == \old(*i) &&
      *j == \old(*j);
  @ ensures \result == 0 ||
                    \result == -1;
  @*/
```

(a) Original annotations of C code

```
requires (C_11 :
  (((C_13 : (\offset_min(i) <= 0)) &&
    (C_14 : (\offset_max(i) >= 0))) &&
   ((C_16 : (\offset_min(j) <= 0)) &&
    (C_17 : (\offset_max(j) >= 0)))));
requires (C_18 : ((r == null) ||
  ((\offset_min(r) <= 0) &&
       (\offset_max(r) >= 0))));
[...]
ensures (C_19 :
  (((C_21 : (i.int_M == \old(i.int_M)))
      &&
    (C_22 : (j.int_M == \old(j.int_M)))
        ) &&
    (C_23 : ((\result == (- 1)) || (\
        result == 0)))));
[...]
```

(b) Annotations translated to Jessie

Figure 3.3: Direct comparison between C annotations and their translations in Jessie

Jessie come from the annotation syntax and are also quite straightforward (`axiomatic`, `requires`, `behavior`, `assumes`, `true`, `false`, `ensures`, `var`).

If we now focus on the translation of the code annotations and compare those from the original C code and those in Jessie (see Figure 3.3), we can see that internal verification functions, in our case `\valid()`, are now translated into the functions that directly check for the memory location of the given variables to make sure that they are allocated within the program (that is exactly the meaning of `\valid()`). There are also functions (like `\old()`, `\result`) that are not translated at all, for they pertain to the current level of abstraction and are resolved only in the next step by the VCC compiler when verification conditions are generated.

An inquisitive explorer can notice that integer values are not represented as structures. That is due to the memory model Jessie implements and uniformly applies for every variable, both primitive and composite ones. More information and discussion on how application memory is modeled in Jessie can be found in Section 4.3.

We would like to point out to the fact that machine memory is represented here by composite variables (structures) containing information about both allocation and the value of every variable used in the original C program. This implies that our memory representation is still based upon machine memory and is not completely logical so far. To enable the generation of verification conditions we need to convert all parts of our software system, including the memory model, into logic that is amenable to SMT proofs. Only then the generation of VCs becomes possible.

3.2.5 Generating Verification Conditions (VCs)

Verification Conditions (VCs) are generated by the compiler of an intermediate proof language. The proof languages are memory-agnostic; they do not implement any explicit memory and store memory states in program variables, both local and global ones. Due to a tool chain split in Frama-C, Jessie is the latest stage in the tool chain that still implements the memory model, whereas VCC does not have any separate pre-proof intermediate language one could easily refer to.

Because neither the intermediate proof language nor the VCs represent executable code, the formal transformation process can, strictly speaking, not be seen as compilation, but rather as code translation. The translation process enables the compiler to add some verification clauses it finds appropriate during this stage and to remove those that contain evident tautology (e.g., expressions that are always true irrespective of the operand values). Therefore, there is no way to easily determine the number of verification conditions *a priori*.

Verification conditions are not just expressed in a memoryless, side-effects free intermediate proof language, but also contain a rather long preamble with (innate logical) functions. Adding a preamble leads not only to a bigger file (refer to Table 5.1), but also increase the number of rules the proof propagation must be matched again. Even though instantiating a theorem is, as a rule, much faster than deducing it, constant pattern matching against a rarely used theorem impacts the overall deduction performance and might lead to timeouts. The trade-off between instantiating and deducing is not easy to find, therefore it is almost impossible to construct a verification tool that would address all possible verification problems with the same performance: narrow-tuning in this case is essential to get verification conditions successfully proved (or disproved) within a reasonable amount of time. We will discuss narrow-tuning as a trait of modern SMT solvers in the next session.

3.2.6 SMT solvers

Satisfiability Modulo Theories (SMT) solvers emerged and became popular only in recent years, mostly due to their extended use in hardware validation. They check satisfiability of a first-order formula with certain functions, variables, and constants, interpreted according to the theories submitted to the solver beforehand in a preamble. These theories, called background theories, typically cover integer and rational arithmetic, bit vectors, and sometimes arrays. Similarly to the intermediate proof languages, there is no memory model attached to SMT solvers and all operation pertaining to memory (pointer arithmetic, reading/writing, concurrency) are encoded using built-in functions (termed theories) and the formulas of first-order logic.

SMT solvers are built on top of existing propositional SAT solvers. The breakthroughs in the latter area were around 2000 with the adoption of binary constraint propagation as the core mechanisms for optimizing SAT solving. Industry-level solvers like MiniSAT or Chaff [Janota 10] account also for the development of some powerful SMT solvers that can be also designated to comply with industry standards for hardware verification SAT solvers.The first SMT solver was introduced in the late 1970s under the name Simplify by the group of scientists around David Dill at Stanford. Confusingly enough, a second group, run by David Detlefs, Greg Nelson, and James Saxe in the course of their ESC projects, developed a solver with the same name. Remaining unbeaten for at least the following 10

years as regards software verification it accounted also for wide adoption of its internal format for SMT problem descriptions (.sx-files) [Detlefs 05].

Modern SMT solvers are developed with tight integration of SAT solvers and decision procedures, which helps to improve their performance. These are Yices, Ergo, CVC3, Fx7, and Z3, to name a few.

Both Frama-C and VCC allow an experienced user to choose an appropriate SMT solver. Jessie has built-in support for four different solvers (Alter-Ergo, Z3, CVC3, Yices) and provides an extensible configuration file to include even more solvers [Marché 10]. It can submit the proof obligations automatically to all of them or just to a pre-defined set depending upon the settings, and can parallelize the calculations. Both Jessie and Why support a rich set of command-line parameters to fine tune the verification process of SMT solvers.

VCC does not support the explicit selection of the SMT prover and is narrowly tuned to the Z3 solver from Microsoft Research, currently using the Simplify file format as the intermediate format to store proof obligations. It is, however, possible to manually set other file formats, for example SMT-LIB 1.2, for output, using a command-line setting for Boogie. The export to this file format seems to be not completed yet, for this file is not yet readable by Z3 itself due to syntax problems.

Chapter 4
4 Memory models

As we have seen in the previous chapter, memory constitutes an important part of every software system and requires reasonable assumptions to be modeled successfully.

More precisely, we do not take into account the actual physical memory and memory management performed automatically or semi-automatically by the operating system, we rather approach the so-called „program memory" – memory structure as it is seen from the running application. Thus, we consider the memory to be flat, beginning at zero and extending to the maximum of addressable space, where only a certain contiguous region in the beginning of this space is allocated to the program and any attempt to access (irrespective of the access kind, both reading and writing) memory outside of this region leads to a general protection fault and is regarded illegal operation. This memory model is consistent with the „protection mode" execution model supported by modern CPUs.

In addition to the constraint that no memory region beyond the upper boundary can be accessed by the application, *memory safety* implies two further constraints: (1) only *properly* allocated data is accessed in the source code, to prevent reading from unallocated memory and (2) allocated data is accessed only through proper accessors, to prevent unintended illegal modification of data or neighboring structures.

We explain both statements in the two following sections.

4.1 Type-safe memory models

Properly allocated data is designated by the fact that the allocated memory chunk has a certain *data type* associated with it. A data type unambiguously defines the kind of the value stored at the given memory location and determines the internal memory representation of these data. It is important here that the data is always accessed as an instance of the pre-defined type, e.g., there is no way to consider this memory chunk as data of any other type or as a control structure. This access paradigm, which excludes the possibility of *memory reinterpretation,* is designated as type safety and is usually enforced by the programming language, called in this case type-safe languages. Since this constraint confines the room for available constructs for programmers, modern general purpose languages usually relax this constraint to so-called strong typing with no memory reinterpretation allowed after its allocation.

Strong typing, enforced at the level of the programming language, is called language-based memory safety. Since C is a classical example of a weakly typed language (see the discussion

in Section 3.2.1) strong typing needs to be explicitly verified for every C program under test.

Despite the fact that C as a language does not enforce type safety, safety still can be enforced by developers using dexterous system architectures together with strict coding conventions (termed *coding discipline*). Since the type-safe assumption about a memory model helps to perform certain types of analysis that are not possible or computationally feasible in case of a more general memory model, many software verification toolkits, including both Jessie and VCC, make this assumption. However, due to the fact that this assumption can be easily violated in C, both toolkits also implement necessary checking mechanisms to make sure that this assumption is a safe one for the code given.

Assuming type safety of C programs allows in particular a computationally cheap separation analysis and inference of certain conditions about memory safety. The authors of VCC proposed the following motto to emphasize this fact: "It's no harder to functionally verify a type-safe program written in an unsafe language than one written in a safe language." [Cohen 09a].

4.2 Aliasing

The term *aliasing* describes the situation where certain memory locations can be accessed through several symbolic labels, not necessarily being of the same data type. In C, aliases can appear if developers use pointers to directly access memory. Depending upon the pointer type or, more precisely, on the fact whether the pointer is a typed pointer (pointing to a certain data type) or an untyped pointer (a void pointer), it can give access either to a single memory unit or to a memory chunk whose size equals to the size of the pointer type. Two pointers are said to be aliases (or aliased) if the memory regions they point to are the same or overlap, otherwise they are not aliased or disjoint [Moy 09].

There are three possible combinations of pointers (see Figure 4.1). Disjoint pointers point to different and disjoint memory regions. Partial aliases are pointers that point to different memory locations but their memory chunks overlap. And complete aliases are pointers that both point to the same memory location and refer to memory chunks that are of the same size. Complete aliases are required to be of the same type, otherwise they are considered partial aliases, even if they have the same size.

The fact that the C language[1] does not provide any built-in mechanism to detect or avoid aliases greatly complicates the analysis of C programs, in particularly static safety checking. Even though aliases are often evident to the developer because they are used purportedly, there is still the risk of non-intentional aliasing. Irrespective of the purpose of aliases, they remain implicit at the source code level. Despite considerable efforts and breakthroughs in software analysis, aliasing remains a major problem in the analysis of C programs.

4.3 Memory models in VCC and Jessie

The two verification toolkits we use exercise different approaches to memory modeling (see Figure 4.2).

[1] The C99 standard defines the keyword `restrict` that can be used to indicate the fact that the given code does not contain any aliases. However, this keyword is rather a declaration of programmer's good intentions to avoid aliasing, because no known conventional C compiler is equipped with alias analysis and can perform the necessary verification of C code during compilation.

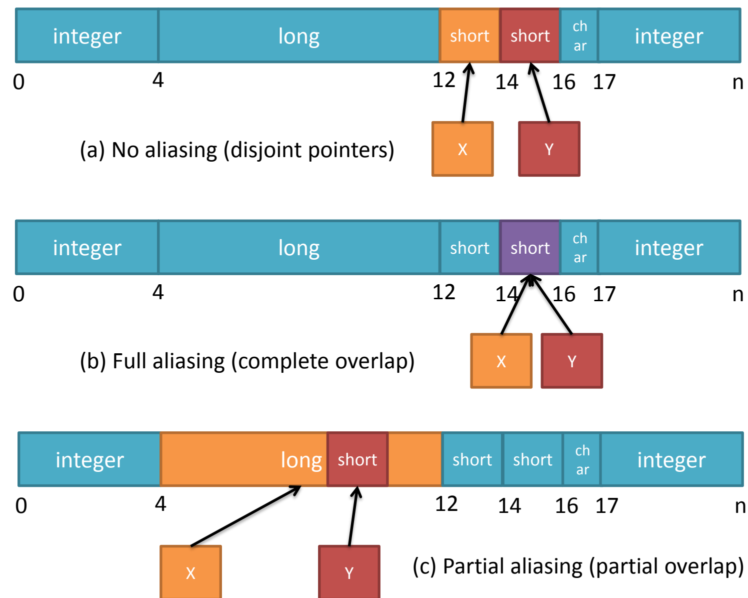

Figure 4.1: Types of aliasing: no aliasing (top), complete aliasing (middle), and partial aliasing (bottom)

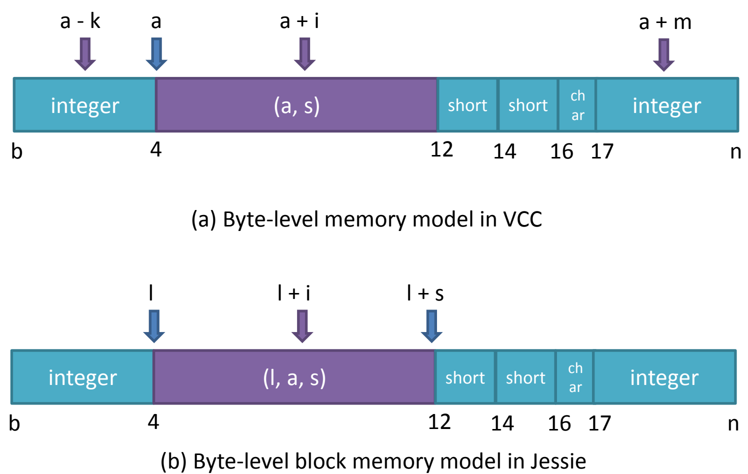

Figure 4.2: Byte-level memory model of VCC (top) and byte-level block memory model of Jessie (bottom)

Generally, memory is represented by a mapping between an address of a memory cell and its contents, where the address ranges from 0 to $N - 1$ and N is the memory size.

VCC (upper subfigure) uses a simple byte-level memory model [Cohen 09c, Moy 09]. In this model, a block of memory is typically represented as a tuple (a, s) of an address and a size, where the pointer contains only the address value (magenta block). The level of abstraction provided by the C program is partly lost in this model, for it is possible to access beyond the bounds of the allocated memory chunk, changing the values of the neighboring locations, which is prohibited by the C standard.

Besides the spatial representation of the application memory, VCC also uses a lock-free concurrency mechanism to associate memory chunks with certain threads and check for violation of concurrent access [Cohen 10c]. This mechanism is based upon a so-called ownership that documents the access rights to the current memory location for the current thread. This memory model, combined with the ownership model in VCC, is narrowly tuned to the concurrency checking of C programs and is only sub-optimal as regards memory safety analysis.

Another approach, which helps to avoid problems with unauthorized memory access to pre-allocated chunks, is the so-called block model, where the complete memory is split into disjoint memory chunks containing no information about their base address. In this case memory blocks are represented as tuples (l, s), where l stands for the label and s for the size and the block is uniquely and solely identified by the label. Every pointer in the program code is then translated into the pair (l, i), as pointing to the certain label with given offset (index), where the condition $i < s$ must be always fulfilled.

Jessie, in turn, is based upon the byte-level block memory model (see Figure 4.2, bottom), which merges the byte-level and block-level models to avoid their limitations [Moy 09, Hubert 08]. Every block is represented by the triple (l, a, s), standing for label, address, and size respectively. Here again, label l fully identifies the memory block, meaning that no two blocks can have the same label. The address value here is just used to identify blocks with potential overlap and is not used in abstracting memory access, which, instead, is encoded using only the label and offset $(l + i)$. This memory organization allows Jessie to effectively update memory locations and to keep track of possible aliases without much overhead.

Besides the region-based model, Jessie also supports a much weaker linear memory model for the case if the assumption about strong typing of the memory locations can not be verified. This model is more flexible as regards the direct memory manipulations of C code, but limits at the same time the analytical capabilities of Jessie and prevents it from performing separation analysis (see Section 5.2.2). This model can be helpful in cases where one needs to perform software verification with the best possible overestimation as regards aliasing, for Jessie then suggests aliasing everywhere.

Considering the semantics of Jessie (see Figure 3.3), there are two built-in functions `\offset_min(x)` and `\offset_max(x)`, which return the first and the last byte of the memory location of a given variable `x`. Thus, the check for variable validity is equal to the check that both start and end byte of the variable lie within the allocated memory region (chunk). The pre-conditions and post-conditions containing the pre-defined function `\valid(x)` are therefore converted into these two functions (see Figure 3.3).

4.3.1 Explicit specification of the memory model in Jessie

There are two different ways to communicate to Jessie that it should use a much weaker (in respect of type safety) memory model. For any given program one can supply the command line parameter `-jessie-no-regions` while calling Frama-C. Any pre-defined or explicitly declared memory models are disregarded in this case and Jessie is forced to add additional assertions to check for aliases and partial memory overlap.

Alternatively, one can explicitly set the memory model used by providing a pragma definition at the beginning of the annotated C code. Jessie supports several pragmas for different types of analysis. Separation analysis (see next section) is controlled by the `SeparationPolicy` pragma whose default value is `regions` (for the strongly typed memory safety model). An alternative value `none` is equivalent to the command line parameter from the previous paragraph and just weakens the assumptions about the memory model of the code under test.

The importance of the memory model and the underlying verification using Jessie will become evident in the next chapter (see Section 5.2.2).

4.4 Separation analysis

Separation analysis is a technique to automatically detect that two given pointers are not aliases of each other [Hubert 08]. In this respect, this term can be seen as a full synonym for the term „alias detection." The latter is, however, a more general term describing all possible measures and techniques in software verification to detect aliases, irrespective of whether separation logic (see next paragraph) was used to deduce possible aliases or not.

Separation analysis often implies the use of separation logic to perform alias detection. Separation logic is a part of the first-order logic for specifying and verifying properties of heap-allocated data. It provides local reasoning, thus allowing compact specifications specifically targeting only memory issues [Botincan 09]. Local reasoning allows the specification to be automatically confined to the memory location that is allocated for the given piece of code rather than dealing with the complete heap.

Every separation logic formula consists of two parts: a pure logic formula Π (not involving heap objects) and a so-called spatial formula Σ (describing the heap).

Separation logic requires a custom-built prover that is built much like a normal SMT prover: it contains the set of predefined rules (background theories) that allow it to modify separation logic formulas and (if possible at all) convert them to pure logic formulas. Those can be then submitted to a normal SMT solver for further verification, because a separation logic prover is not tuned enough to solve them effectively. An example of a separation logic prover is the jStar prover [Distefano 08].

Despite the expressiveness and power of separation logic conditions they require special, narrowly-tuned for separation problems, solvers to be effectively processed. Being not as effective as general SMT solvers, these can have more time-outs and require more discipline and precision in code annotation [Botincan 09].

The traditional approach, taken also in Jessie, is to translate all memory operations (allocation, reading and writing, freeing) into built-in functions dealing with the memory model and just check for violations of certain predefined and hard-coded rules enforcing

memory safety. This approach is not as flexible as the first one, but might be much more performant in real-world applications.

In this thesis we use the term „separation analysis" to denote the algorithms and heuristics used by Jessie to detect aliasing and suggest some contracts for its prevention.

The outcome of the separation analysis are those variables or function pointers that might be aliased. Besides, other types of checks, for example for memory overflow, stack over- and underruns and for heap fragmentation can be also performed using this analysis. The primary goal of these checks is not to detect aliases, but rather to exclude the possibility of unauthorized modification of foreign data, i.e., to enforce memory safety conditions.

Chapter 5
Extended Example

This chapter contains the author's contribution to the topic and demonstrates the approach of verification toolkit interplay for the purpose of alias detection.

The ultimate goal of this chapter is to enable the interplay between the concurrency-oriented VCC framework and Jessie, a deductive verification plug-in for Frama-C, equipped with separation analysis. This goal can be divided in several subsequent steps:

Step 1: To derive an illustrative sample code that could exploit the benefits of separation analysis implemented in Jessie. At the same time, this code should be not correctly analyzed in VCC (Section 5.1).

Step 2: To apply separation analysis and alias detection implemented in Jessie to the code and collect as output both final and intermediate results (Section 5.2).

Step 3: To decide on the level of abstraction one has to look at to see the results of the separation analysis (Section 5.3.1).

Step 4: To (first) manually discover the code sites in the intermediate code that are result of the separation analysis carried out by Jessie/Why (Section 5.3.2).

Step 5: To integrate these findings into VCC's annotations and show how these results can be used to improve software verification done by VCC (Section 5.4).

Step 6: To attempt to generalize the sample code to a more general problem (Section 5.5).

Step 6: To automate the inferring of alias annotations from Why intermediate code and to integrate them into VCC annotations (Section 5.6). In the following sections of this chapter we will try to sequentially implement those steps.

5.1 C code samples

To test how well VCC and Jessie are able to determine whether two different variables could potentially be aliases we design a small piece of C code that is sensitive to aliases in input parameters, and provide a specification that is strong enough to detect possible side effects of this code due to aliasing.

5.1.1 Code characteristics

Before starting to design an appropriate code snippet we derived the principles that should be met by this code, so that we can detect aliases by employing separation analysis for verification:

1. The C code has to be wrapped into a single function.
2. The function must have at least two input parameters whose values are read (to enable aliasing), and one output parameter or return value that is modified within the function body.
3. The expected outcome of the function, available both as a return value and an output parameter, should not be influenced by the possible side effects.
4. If any two input parameters are aliases, the result of the function observably deviates from the correct result and is designated as side effect.

Besides the sensitivity to aliasing, the code should be rather short to serve illustrative goals and portable to be easily converted to another annotation.

5.1.2 The max() function

To meet all these criteria we implemented a simple comparison function in C that takes two input arguments and returns the biggest value (if the values are different, otherwise the first) through an output parameter.

Regarding possible kinds of aliasing (see Figure 4.1), we constrained aliasing to the cases of complete aliases, intentionally disregarding the cases of partial memory overlap of two variables.

To obtain complete aliasing we took a primitive data type, signed integer, as the type for our input argument here. Signed integer as a datatype *per se* does not guarantee the absence of partial overlap because, architecture-dependent, it might occupy either 4 or 8 subsequent bytes for its memory representation. We, however, assume that the memory is reserved in chunks with the minimal chunk size being a multiple of the signed integer size under this architecture. This seems to be a common case on modern hardware architecture and helps to accelerate memory access during arithmetical and logical operations.

The source code of the function is shown in Listing 5.1.

```
1 int max(int *r, int* i, int* j) {
2   if (!r) return -1;
3   *r = (*i < *j) ? *j : *i;
4   return 0;
5 }
```

Listing 5.1: The max() function

Listing 5.1 presents the C function `max()` that takes two input parameters (`i` and `j`) and writes the larger of these two values to the memory location referenced by the only output parameter (`r`). The return value of the function is either 0 or -1 depending upon the availability of the memory referenced by r, in other words whether `r` is `null` or not.

We use integer pointers as function parameters so that we can modify the values of these parameters within the function, thus turning some parameters (namely `r`) into output ones.

According to the specification of C, the input parameters of functions are pushed on the stack before function invocation. Had we used simple integer values instead of pointers, copying their values to stack ("call-by-value") would automatically lead to dis-aliasing of them (because a copy can never be an alias).

To simplify the analysis of the source code and the derivation of its specification we consider only the situation with a single alias when at most two variables that reference the same memory address are considered as aliases irrespective of whether those are the only aliases in the code.

5.1.3 Contracts and annotations

To trigger the static verification of the `max()` function we must decorate it with annotations containing pre- and/or post-conditions or invariants.

The function implements a defensive programming approach as regards the output parameter `r` and checks whether it is not `null` in line 7. The space of possible values for this parameter is thus represented by the null pointer or any valid pointer (to prevent memory protection fault). This can be expressed in form of a pre-condition (see line 2 in Listing 5.2).

To ensure that the ternary operator in line 8 does not produce any exception we would need to check the input parameters for null values. To avoid the overhead due to extensive checks during runtime we can relocate these checks to our specification and make sure that those values are always valid. That adds one more pre-condition (line 1).

Aliases are revealed only if the doubly referenced memory location is accessed for writing, e.g., if one of the aliases is assigned a new value. The only writable variable within the function is the output parameter `r`, which is set to the maximum of the two input parameters. Therefore, we consider only the cases where the output parameter `r` is aliased with any of the two input parameters.

To detect the alias situation we should look for the most common effect imposed by aliasing. Let two variables be aliases, then the value change of the second, unrelated variable is updated automatically whenever the first variable is assigned the new value. It is thus possible to design a post-condition contract using the annotation function `old()`. The `old()` function (sometimes also referenced as `pre()`) returns the original value for every variable before the invocation.

The resulting annotated code of the `max()` function is presented Listing 5.2:

```
1  /*@ requires \valid(i) && \valid(j);
2    @ requires r == \null || \valid(r);
3    @ ensures *i == \old(*i) && *j == \old(*j);
4    @ ensures \result == 0 || \result == -1;
5    @*/
6  int max(int *r, int* i, int* j) {
7    if (!r) return -1;
8    *r = (*i < *j) ? *j : *i;
9    return 0;
10 }
```

Listing 5.2: The max() function annotated in ACSL for Jessie

5.2 Alias detection with Jessie and VCC

VCC does not implement separation analysis, however direct comparison between two pointer variables can help answer the question whether those two pointers are aliased or not. This comparison should be, however, explicitly defined in code annotations, for by default, VCC consider all pointers as possible aliases. In this respect the benefit of separation analysis would be to find out the minimal set of pointers that should be distinct so that the conditions in the annotated code hold.

5.2.1 Separation of memory regions in Jessie

Since the source code of the Jessie program is rather long we do not provide the complete listing. You can find it in Appendix A.2 on page 49.

If we now take our annotated program from Listing 5.2 and run the verification of this code using Frama-C and Jessie, we will see that all contracts have been successfully checked disregarding the fact that some parameters in the function signature may be aliases.

Proof obligations	Alt-Erg 0.9
Function max Safety	✓
1. precondition	✓
2. pointer dereferencing	✓
3. pointer dereferencing	✓
4. pointer dereferencing	✓
5. pointer dereferencing	✓
6. pointer dereferencing	✓
7. pointer dereferencing	✓
8. pointer dereferencing	✓

Figure 5.1: Verification conditions for Listing 5.2 shown in gWhy

This might seem puzzling at first glance, but there is a simple explanation why Jessie does not reveal any potential problems with memory safety in our sample code: we have just annotated our code with contracts, but said nothing about the memory model. By default, Jessie assumes a strongly typed (region-based, see Section 4.3) memory model for C programs by default. We have discussed in Section 4.3.1 the pragmas that allow one to explicitly select a memory model in Jessie .

Listing 5.3 shows that we specify a weaker memory model explicitly to reach the expected behavior:

```
#pragma SeparationPolicy(none)
/*@ requires \valid(i) && \valid(j);
  @ requires r == \null || \valid(r);
  @ ensures *i == \old(*i) && *j == \old(*j);
  @ ensures \result == 0 || \result == -1;
  @*/
int max(int *r, int* i, int* j) {
  if (!r) return -1;
  *r = (*i < *j) ? *j : *i;
  return 0;
}
```

Listing 5.3: The max() function with annotations and pragma

Trying to verify this code in Jessie (in automated non-interactive mode) now meets our expectations: it fails (see the console output on Listing 5.4), because the post-condition cannot be verified given the weaker memory model.

Generally speaking, this error with aliases can be seen as an overapproximation, for the `max()` function implements the correct behavior if the input parameters are not aliased with the output parameter. To make sure whether this potential problem really comes into play in a software system we need to analyze the complete system the given code represents only a small part of. In the next section we will extend our code snippet to a tiny, but complete software system.

```
Running Simplify on proof obligations
(. = valid * = invalid ? = unknown # = timeout ! = failure)
simplify/max-none_why.sx : ?..?........ (10/0/2/0/0)
```

Listing 5.4: Output of the max() function verification with Jessie, pragma SeparationPolicy is set to none

5.2.2 Memory separation analysis in the scope of a complete C program

To judge whether aliasing really takes place during invocations of the `max()` function we need to extend our previous listing so that it becomes a complete C program. This is done by adding an entry point (that traditionally is the void or integer `main()` function), a variable declaration and assignment, and one (or more) `max()` function invocations.

```
1  #pragma SeparationPolicy(none)
2  /*@ requires \valid(i) && \valid(j);
3    @ requires r == \null || \valid(r);
4    @ ensures *i == \old(*i) && *j == \old(*j);
5    @ ensures \result == 0 || \result == -1;
6    @*/
7  int max(int *r, int* i, int* j) {
8    if (!r) return -1;
9    *r = (*i < *j) ? *j : *i;
10   return 0;
11 }
12
13 /*@ requires \valid(a) && \valid(b);
14   @*/
15 int main(int *a, int *b) {
16  int c;
17  max(a,a,a); // function invocation with aliases
18                 // -- parameter sets are not disjoint
19  /*
20  max(&c,a,b); // all three different parameters - disjoint sets
21  max(&c,a,a); // two groups of parameters - sets are still disjoint
22  */
23  return c;
24 }
```

Listing 5.5: The complete C programm with declaration, implementation, and invocation of the main() function. The invocation has some aliased parameters.

The code provided in Listing 5.5 is an example of such a short C program. For simplification reasons we have just a single function invocation within the body of the `main` function, and as one might easily see in line 17, this invocation contains parameters (`&c`) that are complete aliases. It is not surprising that Jessie fails to prove this C program as well, for it requires the regions to be disjoint and the sets `&c` and `&c` are equal and thus not disjoint.

A slight modification of the code above, namely the use of either all three different parameters (line 20 in Listing 5.5, commented out) or at least different parameters for input and output values (line 21, same listing) has the consequence that the program becomes verifiable. The post-condition for `max()` guaranteeing the absence of possible side effects of aliases is proved correct now. For such a small program, one can see with a naked eye that there is no other invocation in the main body, for bigger software systems, the help of automatic tools becomes really essential.

As one may see, Jessie proves that the invocation parameters are disjoint using separation analysis of memory regions prior to function execution. This check is implemented as an additional „inferred" assertion, which is automatically added to the verification conditions to be proved. If the assertion can be proved, this information (that there is no aliasing of the parameters) propagates further to the function call site and helps decide whether the post-conditions hold or not.

5.2.3 Adapting the example for VCC

As already mentioned in Chapter 4, VCC implements a byte-level memory model that seems to be less powerful as regards the separation analysis. It can be likened to the `none` setting of the normally region-based memory model in Jessie, where every two pointers are considered to be aliases unless the opposite is assumed (or asserted). We now adapt our example from the previous section to VCC annotations to see how this prover and compiler deals with our contracts.

Before all, we need to translate the ACSL annotations to the VCC format; the remaining code is a pure C code and does not require any modification.

Considering the fact that VCC is narrowly tuned for concurrent analysis of C programs we need to supply some additional annotations (`thread_local()` etc.) to pacify this type of analysis. The resulting code is shown in Listing 5.6.

```
1  #include <vcc.h>
2  int max(int *r, int* i, int* j)
3    requires (*i && *j && *r)
4    writes (r)
5    ensures (*i == old(*i))
6    ensures (*j == old(*j))
7    ensures (result == 0 || result == -1)
8  {
9   assume(thread_local(i));
10  assume(thread_local(j));
11  assume(thread_local(r));
12  if (!r) return -1;
13  *r = (*i < *j) ? *j : *i;
14  return 0;
15 }
```

```
16
17 int main(int *a, int *b)
18  requires(a && b)
19 {
20  int c;
21  max(&c,a,a);
22  return c;
23 }
```

Listing 5.6: Complete C programm with max() function ported to VCC

In this implementation for VCC we do not examine separately the cases where some parameters are aliases but directly start with an indicative example where the parameter sets are disjoint and thus cannot be aliases.

Unlike Jessie, VCC requires us to explicitly mark the variables that will be changed during execution as `writes(vars)`, so that they are checked to have been allocated in memory (e.g., no null pointer are allowed for these variables). That explains why we do not declare the variable `c` as an integer pointer, but rather as a local integer variable and provide just a reference to it in our `max()` function. This automatically ensures that `c` has been allocated and can not be a null pointer.

5.2.4 Modular approach to static source code analysis in VCC

As we saw in Section 5.2.2, Jessie is capable of determining of whether a function invocation contains aliases or not by analyzing parameters sets.

Running VCC to verify the code from Listing 5.6 leads to post-condition failure despite the fact that the function invocation contains no aliases and should not offend any pre- or post-condition. The output of VCC is shown in Listing 5.7.

```
C:\VCC>vcc max-main.c
Verification of max failed.
C:\VCC\max-main.c(15,2) : error VC9501: Post condition '*i == __old(*i)'
    did not verify.
C:\VCC\max-main.c(6,14) : error VC9599: (related information) Location of
    post condition.
Verification of main failed.
C:\VCC\max-main.c(22,2) : error VC9502: Call 'max(&c,a,a)' did not verify.
C:\VCC\max-main.c(4,15) : error VC9599: (related information) Precondition
    : '*i'.
Exiting with 3 (2 error(s).)
```

Listing 5.7: Output of VCC for the code in Listing 5.6

This result can be explained by the modular approach to verification VCC takes in order to check source code, as illustrated in Section 2.5. Just to briefly recall the main idea behind this modularity: Every function is initially checked for contract compliance using solely its implementation and with no additional information from outside to be passed to it. As soon as the function contract is proved, it is taken for granted for any further verification without re-verification of the function (to reach better performance).

Level of abstraction	Filename	Size (in Kb)	Lines of Code (LOC)
C code annotated with ACSL	max-main-none.c	0.5	18
Jessie	max-main-none.jc	2	86
Why	max-main-none.why	28.5	795
SMT-LIB 1.2	max-main-non_why.smt	185.7	4807

Table 5.1: Comparison of the source file in different (proof) languages

In our case, we could have side effects if aliases are passed as parameters to the function and we want to exclude this possibility. What we want is that our contracts become verifiable in VCC and for that, we need to extend the pre-condition of our function and make it stronger. We now look back to Jessie in an attempt to find the correct formulation of this pre-condition.

5.3 Inferring alias contracts from Jessie and Why

5.3.1 Determining the level of abstract for annotation inference

Going from the original, annotated C code downstream to the SMT-provers rises the level of abstraction from the original code and leads to a dramatic increase in the file size, which decreases the readability and thereby impedes the human analysis of these files. The comparison between file sizes and lines of codes (LOC) at different abstraction levels in the tool chain is given in Table 5.1.

Because of the dramatic surge of the code length at the highest level of abstraction, we had to confine our analysis to the two intermediate levels represented by Jessie and Why and to try to automate annotation inference as soon as the source code analysis is done by Jessie.

As we know from Section 3.2, the translation from annotated C to Jessie does not change the original contracts but just performs the abstraction of the source code with memory access operations translated to reading and writing of typed variables. Therefore, the Jessie code contains nothing new compared to the original contracts and we have to begin our analysis with the code in Why, especially the VCs generated by the Jessie2Why compiler.

5.3.2 Using Why code to infer annotations for VCC

Why programs provide more interesting opportunities to look into how the verification conditions are built-up, because they do not have any memory model attached to the program code and every annotation statement about memory location (reading, writing, etc.) is encoded explicitly.

Given the complexity of Why code it is easier to track and analyze proof obligations using graphic tools. Already shown on Figure 5.1, gWhy is not only capable of showing the verification conditions and of running provers, but also provides the possibility to look at the Why source code for the selected assertion (highlighted in the figure).

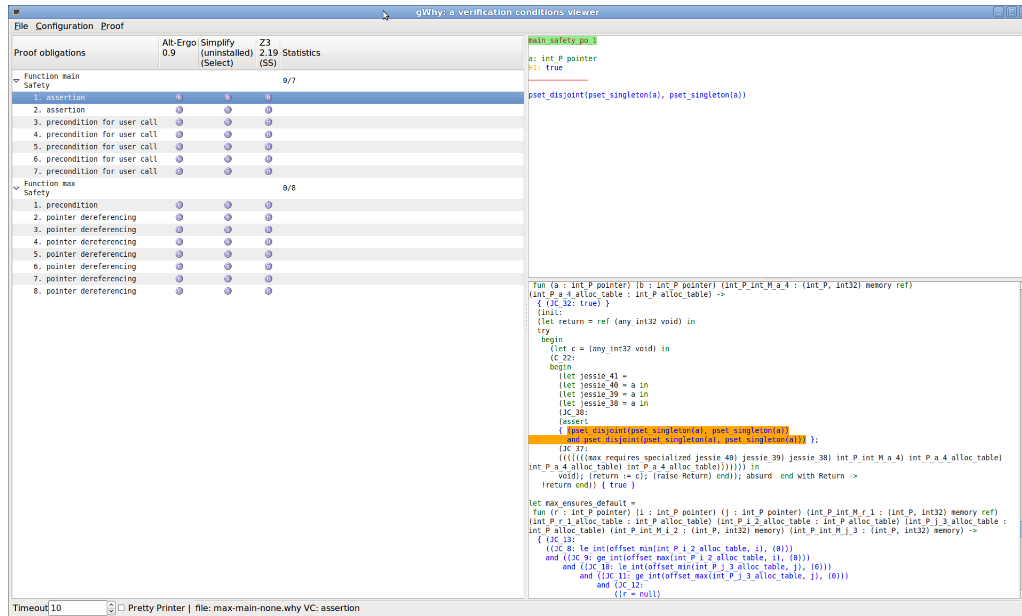

Figure 5.2: Output of gWhy with two assertions inferred by separation analysis (left) and the underlying Why code (right)

Two assertions that were generated by Why as the result of separation analysis are shown at the top of the „Function main Safety" section (the top assertion is highlighted); the underlying Why code is shown in the subwindow to the right. You can also see this code in Listing 5.8.

```
1  assert
2        { (pset_disjoint(pset_singleton(a), pset_singleton(a))
3          and pset_disjoint(pset_singleton(a), pset_singleton(a))) };
```

Listing 5.8: Inferred verification conditions in Why

Both conditions are formulated the same way and use the Why built-in function `pset_disjoint(a,b)` (defined in the preamble), returning true if arguments have disjoint memory locations and thus are not aliased. In the code of `main()`, the invocation of the `max()` function takes place with three parameters. The first assertion here makes sure that the first and the second arguments are not aliased; the second assertion checks the same for the first and the third arguments (output and input parameters). Non-aliasing of input and output parameters is a prerequisite for side effects, namely any change of the value of an input parameter, not to manifest. This change is explicitly prohibited by the specification and would lead to the violation of post-conditions and verification failure. In this case, both assertions fail, because no variable can not be disjoint with itself.

Trying to reverse engineer the Why code we need to assure that the first argument of the `max()` function is not aliased with the second or third argument. Since in our case these arguments are pointers to primitive data types, we can just use the inequality of pointers to ensure disjointness, which would also ensure that these pointers are not aliased. Therefore we can formulate an appropriate pre-condition using the basic syntax of VCC annotations (see Section 5.4)

5.4 Using inferred annotations to strengthen the specification in VCC

After we have seen how the disjointness of the function parameters was inferred and smartly used in the verification process by Jessie, we now turn our attention to the Verifying C Compiler (VCC) from Microsoft Research and try to integrate this condition to its verification environment. This extension of the original VCC annotations would allow us to strengthen our pre-conditions for the `max()` invocation and thereby help VCC check pointers for aliasing even without explicit separation analysis.

The missing pre-condition seems to be rather obvious in this case and requires no aliasing between the output and any of the input parameters. The extended source code became immediately verifiable in VCC and is shown in Listing 5.9.

This additional annotation can be just guessed by a specification engineer, and for the presented case that might even seem trivial. However, for large and complex systems the „guessing" may not function anymore – exactly for that reason we tried to let Jessie „guess" the missing pre-condition and then infer its annotation.

```
1  #include <vcc.h>
2  int max(int *r, int* i, int* j)
3   requires (*i && *j && *r)
4   requires ((*r != *i) && (*r != *j)) // <-- additional pre-condition
5   writes (r)
6   ensures (*i == old(*i))
7   ensures (*j == old(*j))
8   ensures (result == 0 || result == -1)
9  {
10  assume(thread_local(i));
11  assume(thread_local(j));
12  assume(thread_local(r));
13  if (!r) return -1;
14  *r = (*i < *j) ? *j : *i;
15  return 0;
16 }
17
18 int main(int *a, int *b)
19  requires(a && b)
20 {
21  int c;
22  max(&c,a,a);
23  return c;
24 }
```

Listing 5.9: The C program with extended VCC annotations

5.5 Generalizing the sample code

As we could see in the previous section, annotation inference helps us to derive a correct specification (or strengthen a weak one) in case of the `max()` function, which was devised

using the characteristics from Section 5.1.1. Now we try to extend those principles to cover other possible alias situations similar to this one.

1. *The C code has to be wrapped into a single function.*
   We can provide pre- and post-conditions only for a complete, self-described code block that best is implemented as a single function in C. Functions can contain both input and output parameters and a return value. Output parameters are those that are modified within the function body, whereas input parameters must retain their original values after the function was executed.
   Wrapping the code in a single function is then allows us to use contracts to explicitly specify what parameters of the function we consider input and output parameters.

2. *Aliases are possible only if we use two (different) variables of the same primitive data type.*
   We intentionally confine aliases to primitive datatype variables, so that we only look for complete aliases. In case of complete aliasing, it is easy to prove that two variables are not aliased by proving that pointers to these variables are not the same. In case of more complex data types (structures, classes, etc.), we need to prove whether memory locations occupied by instances of these types are disjoint, and that requires support for separation logic at the level of a proof language or memory model, which we do not have in case of VCC.
   By „different variables" we mean the variables that represent different entities in the program code. These variables are not necessarily aliased, but this possibility cannot be excluded.

3. *The variables must be passed as references to the code under test.*
   Because the only way to pass parameters during function invocation is „call-by-value", i.e., every parameter supplied in the invocation is copied to the stack and then popped to initialize a local variable in the function body, we need to provide references to our original memory locations of both variables, giving the function the ability to modify those.
   By „references" we mean pointers, because C does not support any other types of references (in contrast to C++).

4. *Of the two parameters passed to the function under test as references, one must be (re)assigned within the function code and the second one must retain its original value.*
   As already mentioned in the second point of this list, we cannot use separation logic in our specification to check for aliasing. The only way to detect aliasing is to make its side effects manifest and check for these side effects. The main side effect of aliasing is the unexpected value change of the aliased variable(s). Hence, we need to (re)assign one of the variables (we term it output parameter) we passed as reference within the function body. At the same time, the second parameter (input parameter) we pass to the function must not be affected by this assignment, i.e., it must retain the same value it had before the assignment. For the sake of simplicity we do not wrap the single assignment in assertions, but rather integrate the checks for variable invariance into the function specification. That means, that the second parameter must not be changed in the function, so that in the situation with no aliasing the `input_parameter = \old(input_parameter)` holds.

To summarize all the points above in a single statement, we need a function (1) taking two (2) typed pointers (3) and (re)assigning the value one of the pointers is pointing to (4); the value of the memory unit the second pointer pointing to must remain unchanged (5).

Due to the modular nature of software verification the definitive answer about aliasing in the system is decidable only if we have the complete source code of the system, and can track the calling sequence and check that the inferred or strengthened annotation holds in all possible program runs. The pseudo code of the generalized program structure is presented in Listing 5.10.

```
/*@ [[inferred: requires *a != *b;]]
  @ ensures *a != \old(*a);
  @ ensures *b = \old(*b);
  @*/
void fun(int *a, int* b) {
  *a = *;
}

void main() {
 int a,b; // variable declaration
#ifdef aliases
 fun(&a,&a); // function invocation with aliases
#elif noaliases
 fun(&a,&b); // function invocation without aliases
#endif
}
```

Listing 5.10: Generalized sample code for alias detection, pseudo-code

5.6 Automated annotation inference

To perform automated annotation inference we implemented a console application in C# called Jessifier. This simple Why parser detects verification conditions from Listing 5.8 (page 38). The conditions, once detected, are then parsed and matched against the C code with VCC annotations to find the function declaration(s) that might have aliasing problems. Depending upon the options used for starting the console application, it might either suggest certain changes or directly try to turn verification conditions into annotations and insert them at the correct location in the source code annotated for VCC. The source code for the parsing function and of the regular expressions used to extract the necessary information from the original files is provided in Appendix A.3 (see page 57).

The console application can perform alias inference in two different modes. The inference mode (option „i") is the more advanced and invasive mode; the application not only detects possible aliases, but also tries to generate the missing contracts and insert them into the source code. The more moderate suggestion mode (option „s") also tries to generate the alias contracts, but does not insert them into the code file. This mode can be used for alias inference in complicated cases where function invocation contains several input and output variables or there are several different functions with missing alias contracts (see Section 5.6.2).

5.6.1 Implemented heuristics

The different single steps of the implemented algorithm are presented on Figure 5.3. The inference algorithm uses two source code files: the Why file generated for the

given C source code file (by default located in `source_code_filename.jessie\Why\source_code_filename.why`) and the file prepared for verification with VCC, i.e., containing the source code annotated using the syntax of VCC. This file must have the same name as the original C source code file, appended with `_vcc` (for instance, `max_function_vcc.c`). These two files are schematically represented in Figure 5.3 using blue (for Frama-C and Why) and red (VCC) colors. The outcome of the algorithm is either directly the modified file with VCC annotations or just suggestions for the user how these annotations can be improved.

Parsing starts with the detection of the verification conditions that contain the disjoint clause. These conditions are generated by Why only if there is an aliasing that must be explicitly resolved, and detection of these clauses indicates that some functions in the source code are sensitive to aliasing of its parameters. In Listing A.2, these clauses are located in lines 179 and 180.

The number of clauses connected with „or" indicates possible combinations of aliased parameters in a single function invocation. The clause requires disjointness of two function parameters, where one of the parameters is an input and the second one is the output parameter. Should these parameters be represented by different program variables, the clause `pset_disjoint(pset_singleton(a), pset_singleton(b))` is turned into a tautology, because according to the region-based memory model (see Section 4.3), distinct program variables are always located in disjoint memory locations. Due to pre-optimization of the generated verification conditions, this tautological condition is then excluded from the verification by Why.

Supplying the same variable for both parameters leads to the generation of the following clause: `pset_disjoint(pset_singleton(a), pset_singleton(a))`. Here, `a` is just an arbitrary variable, which can be substituted by any other variable name. Again, for the same reason as in the previous paragraph, program variables with identical names are considered to be one variable and since memory location can not be disjoint with itself this clause is always false. Verification conditions that are always false, in contrast to tautologies, are very important in verification and are always added to the final set of generated verification conditions. This explains the fact why we see these disjoint clauses only in the case if we supply explicit aliases as parameters for main functions.

As soon as disjoint clauses are detected, the program tries to reconstruct the function invocation that triggered the generation of this clause. The disjoint clause is located in the Why code at the site of function invocation, which is not so easy to be recognized. In Listing A.2, the invocation takes place in the lines 173 – 176, where the line 173 opens the block for conditions required for function invocations and the lines 174 – 176 contain the parameters that are supplied to the function. The exclamation mark here serves as unary operator to get the value for a given variable name („*" for pointers in C). The name of the function is, however, not directly indicated in the invocation. Nonetheless, it appears in the lines following the disjoint clause, these are lines 181 and 182. These lines implement the matching of function variables against the function contract; the name of the function „`max`" can be then directly read from this location. This step concludes the analysis of the why file, the rest of inference is done using the pre-annotated VCC code (see Listing A.1).

With the knowledge about the function name and its parameters we can reconstruct the invocation exactly how it appears in the C source code and check if this invocation really exists in the source code at the given location or not. This step helps us to distinguish between aliasing in the source code and possible aliasing of ghost variables used in the

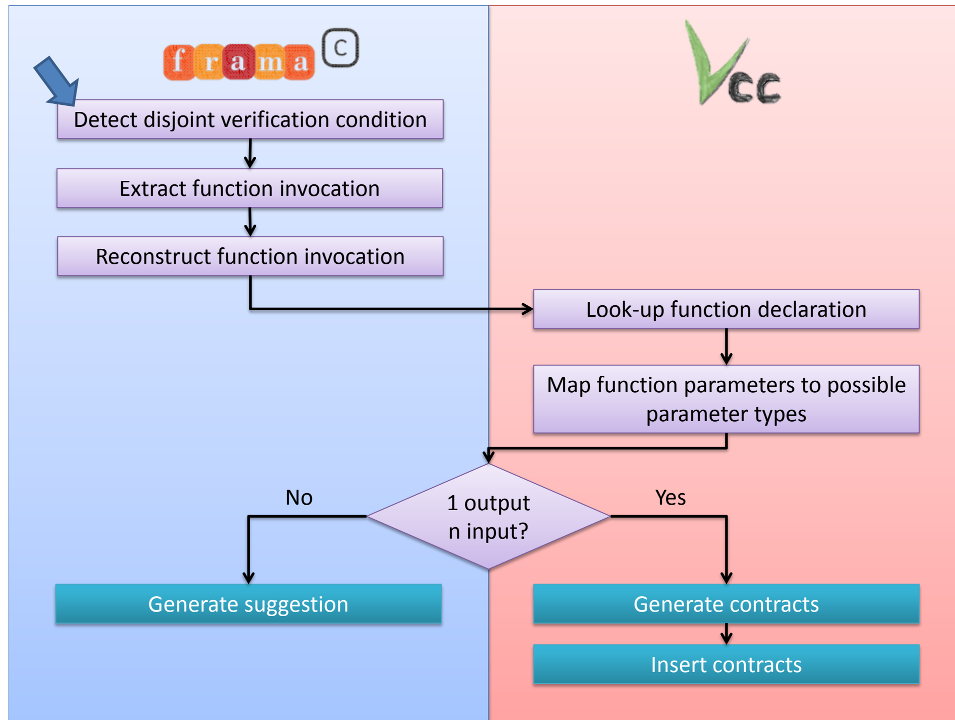

Figure 5.3: Algorithm for automated contract inference for VCC from Why code

annotations. From this step on we use the source code annotated for VCC to continue with the inference of annotations.

Due to the fact that C does not support optional parameters or parameter overload, a function declaration can be easily derived from its invocation. We first perform a short look-up for all possible function declarations and then see how their signatures fit the function invocation. As soon as a function declaration with identical name and number of parameters is found, we extract it, including its annotation, and constrain our analysis to this block only. In case of Listing A.1 that will be lines 2 – 7. From these lines, we infer input and output parameters by analyzing the annotations. Output parameters must be changed within the function body and duly annotated with `writes (param_name)` in VCC (line 4). Input parameters, contrarily, must remain unchanged and annotated using the `old()` function (line 5 and 6).

Since the disjoint clause ensures that input and output parameters are not aliased, we can unambiguously map these clauses to function parameters only if there is only one input or one output parameter. If this were not the case, we can not be sure which of the several parameters must be disjoint from any of the other group of parameters. In this case, the program will generate a suggestion for the user, informing him or her that disjoint clauses were found in the code but parameter mapping was not more possible due to the lack of necessary information at the level of Why. Otherwise, the program tries to generate the inferred annotations and insert them directly into the VCC file. The resulting text file is then output in console and written to the text file.

In Appendix A.3 you will find some excerpts from the source code of the console application: the function `DoParse()` implements the complete logic of the algorithm depicted in Figure 5.3. The set of regular expressions that was used to detect and extract the required information from the source code files also is provided (Listing A.4).

5.6.2 Application domain of the inference algorithm

The algorithm described in the previous section can infer missing alias contracts in arbitrary C programs that are amenable to separation analysis performed by Jessie. In general, this analysis is only possible if the complete C program is written type-safe (e.g., without intentional aliasing of variables or partial overlap in memory locations for lengthy objects), otherwise the region-based memory model of Jessie can not be used for the program verification, and separation analysis can not be done.

The programs must be pre-annotated with ACSL for Jessie and VCC syntax for VCC. Both annotations must be complete enough to enable distinction between input and output parameters, for this is the most crucial point in the contract inference. The algorithm implementation is able to detect incomplete annotations and provides suggestions how these can be amended.

The contracts can be automatically generated only if there is only one input or output parameters – this is the only case that can be unambiguously solved. For other cases suggestions are generated that can guide the user towards missing contracts even though they can not be generated fully automatically.

Due to the technical limitation of Jessie verification that only a single source code file can be processed in one session, there is a possibility that several functions may have missing alias contracts. In this case, the algorithm suggests several contracts for every function, but does not try to insert the generated contracts into the source code. It is recommended in this case to use the suggestion mode.

Chapter 6
6 Discussion and conclusions

6.1 Difficulties of inferring an annotation

Despite certain success in the annotation inference using the Why output of the Jessie2Why compiler tool, there were several considerable problems the author had to overcome to make this inference possible.

Due to technical limitations, verification tools for separation analysis (Jessie and Why) can be compiled only under native Linux environment (the compilation in Cygwin, – a minimal Linux environment ported to Windows, and using MinGW both failed), whereas VCC runs only on Windows platform. This separation seriously impedes the analysis of the source code and of annotation inference, because the source files have to be manually copied from one system to another. We set up a virtual machine running Ubuntu 10.4 hosted by Windows 7 Professional to perform all types of Frama-C analysis together with VCC.

An important conceptional problem is the encapsulation of separation analysis within the Jessie tool. The input file here is the Jessie code that is just pre-processed C code with unified annotations and several direct translations of the annotations and of the conditions (for example, the `\valid()` function is substituted by the explicit check of whether the memory pointed to is accessible or not). The output of the compiler is the verification-ready Why code that contains the complete set of verification conditions, both derived from the code annotations and added as a result of separation analyses carried out by Jessie. After this step, the code is agnostic of any memory model (for memory locations are converted into variables) and there is no way to distinguish between the derived conditions, stemming from annotations, and those that were generated (inferred).

Jessie has a built-in optimizer that checks the verification conditions before they are translated into Why. One of the most common optimization here is to exclude those conditions that represent tautologies, i.e., logical formulas that are always true irrespective of the values of their operands. These tautologies often occur in case of the region-based memory model, for this model assumes that two variables always reference disjoint memory units unless they are explicitly assigned to the same location. In this respect, the memory safety conditions that are generated and verified during separation analysis, appear in the Why output only if these conditions are not met (that is, they are not tautologies anymore). That limits our inference to the test cases where Jessie detects aliases. If we take the generalized sample code from Listing 5.10, the annotation in the very first line is inferred only if the „aliases" macro is defined and the function `fun()` is called with

aliased parameters, because a variable can not be disjoint with itself (see Section 5.3.2, Listing 5.8). In case if there is no function invocation with aliased parameters, if only the macro „noalias" is defined, the disjointness is considered to be a tautology and the results of separation analysis do not appear in Why output.

Other possible problems, like the complexity of the source code to mine for annotations, can impair the human readability and obstruct manual annotation inferring, thereby supporting the argument for automation of this task.

6.2 Related work

In this work, we make an attempt to improve software verification by using several tools, built upon different architectures and implementing different types of analysis.

The fact that neither VCC nor Frama-C are but monolithic and use subsequently several different tools internally to perform software verification makes them good candidates for the extension by combining them with different, sometimes more powerful implementations, even if they run on different platforms. The most common target for such „substitution" are SMT provers: Jessie brings already built-in support for multiple SMT provers, but VCC[1], with its ongoing support for the SMT-LIB format, now can also target several SMT provers, even though this coupling might require more scripting than in case of Jessie.

The classical way to extend the analytical power of verification is to use semi-automated verification using proof assistants instead of fully automated verification with SMT solvers. Frama-C and Jessie traditionally target Coq [Barras 08, Barthe 06], whereas VCC provides support for HOL/Isabelle [Nipkow 11, Böhme 10, Böhme 08]. Other parts of the verification chains are more difficult to substitute, though both tools provide extensible architectures for third-party plug-ins that allows one to use other C pre-processors and memory models for certain types of analyses [Cohen 10c].

Two works are known that use the extensibility feature of VCC and provide some custom implementations. [Botincan 09] extended VCC with a separation logic analysis plug-in using their own preprocessor for the C language, a custom-implemented memory model, and the separation logic prover jStar. The second extensions was implemented by the author of VCC itself to provide support for assembler code of the annotated C routines [Cohen 10c].

As regards separation analysis, there are numerous different implementations both of complete verification toolkits [Appel 07, Hubert 07] and of standalone separation logic solvers requiring some kind of pre-processed and annotated code to work [Distefano 08, Berdine 05, Nguyen 07].

Despite the publication by [Botincan 09] about separation analysis for VCC, the work was discontinued and the implemented solution is no longer compatible with the current implementation of VCC and jStar [personal communication]. This means, that currently there is no way to perform separation analysis directly within the VCC tool chain and the solution proposed in this work, despite all the problems described in Section 6.1, is the only possible way to carry out separation analysis for the VCC toolkit.

[1] It is actually Boogie, which is a separate project within Microsoft Research, not VCC directly, that gets the support to output the generated verification conditions in SMT-LIBv2 format.

6.3 Future work

In the current work we demonstrated how VCC verification can benefit from analysis performed by another verification tool, Jessie. No doubts, there are also cases in which VCC could provide valuable results for Jessie or generally for Frama-C. The interplay of both verification systems leads to their reciprocal strengthening and saves the work of re-implementation for the verification approach already implemented on another platform. Nonetheless, the fact that both systems target (and currently require for compilation) different platforms negatively influences the ease and effectiveness of their interplay.

It is advisable and expected that the verification tools become more platform-agnostic, so that they can be easily combined or even directly pipelined for certain types of analysis. First steps in these direction could be already seen in the current development: VCC and Boogie go towards SMT-LIBv2 as default format for internal dataflow between them and Z3 (that requires good support for this standard on both sides). The release cycle of Z3 now also contains releases compiled for Unix platform that can be directly used on POSIX-compatible systems. The ongoing improvement in CygWin and MinGW environments will definitely bring us further as regards the compilation and execution of Jessie and Why under Windows platform.

Despite the fact that the inference of annotation was implemented as a standalone command-line utility „Jessifier", the integration of both tools still requires much attention from the user and includes at least one manual step (files processed by Why must be copied to the system with VCC installed). A tighter integration of the analysis done by Why in the architecture of VCC using its plug-in API would be also a logical continuation of the current work and would increase the number of plug-ins available for VCC .

Taking into account the importance of separation analysis for general software verification and steadily growing demand for verification tools on the software market a proper independent implementation of it for the VCC toolset would be also prudent. Using separation logic here as the most general approach would allow one to avoid the re-implementation of an already existing analysis approach from Jessie on VCC and same time would equip VCC with strong reasoning about allocated memory units. The previous, now already obsolete implementation of this approach [Botincan 09] had performance problems with separation logic solver. Exactly here one could employ the latest achievements by the Microsoft Research group working on Z3 SMT-solver and use their knowledge and experience to implement an efficient separation logic solver on top of it.

Appendix A

A Appendix

A.1 Sample C program annotated for Jessie and VCC

```
1  #pragma SeparationPolicy(regions)
2  /*@ requires \valid(i) && \valid(j);
3    @ requires r == \null || \valid(r);
4    @ ensures *i == \old(*i);
5    @ ensures *j == \old(*j);
6    @*/
7  int max(int *r, int* i, int* j)
8
9
10
11
12
13 {
14
15
16
17  if (!r) return -1;
18   *r = (*i < *j) ? *j : *i;
19   return 0;
20 }
21
22 int main(int a, int b)
23
24 {
25   int c;
26   max(&a,&a,&a);
27   return c;
28 }
```

Sample C program annotated for Jessie and used as input during annotation inference

```
1  #include <vcc.h>
2
3
4
5
6
7  int max(int *r, int* i, int* j)
8    requires (*i && *j && *r)
9    writes (r)
10   ensures (*i == old(*i))
11   ensures (*j == old(*j))
12   ensures (result == 0 || result == -1)
13 {
14  assume(thread_local(i));
15  assume(thread_local(j));
16  assume(thread_local(r));
17  if (!r) return -1;
18  *r = (*i < *j) ? *j : *i;
19  return 0;
20 }
21
22 int main(int *a, int *b)
23  requires(a && b)
24 {
25  int c;
26  max(a,a,a);
27  return c;
28 }
```

Sample C program annotated for VCC and used in annotation inference

A.2 Jessie code of the sample C program

```
# IntModel = bounded
# InvariantPolicy = Arguments
# SeparationPolicy = Regions
# AnnotationPolicy = None
# AbstractDomain = Pol
# SeparationPolicy = None
axiomatic Padding {
  logic type padding
}
type int32 = −2147483648..2147483647
type int8 = −128..127
tag int_P = {
  int32 int_M: 32;
}
type int_P = [int_P]
tag char_P = {
  int8 char_M: 8;
}
type char_P = [char_P]
tag void_P = {}
type void_P = [void_P]
int32 max(int_P[..] r, int_P[..] i, int_P[..] j)
  requires (C_11 : (((C_13 : (\offset_min(i) <= 0)) &&
                      (C_14 : (\offset_max(i) >= 0))) &&
                     ((C_16 : (\offset_min(j) <= 0)) &&
                       (C_17 : (\offset_max(j) >= 0)))));
  requires (C_18 : ((r == null) ||
                     ((\offset_min(r) <= 0) && (\offset_max(r) >= 0))));
behavior default:
  assumes true;
  ensures (C_19 : (((C_21 : (i.int_M == \old(i.int_M))) &&
                     (C_22 : (j.int_M == \old(j.int_M)))) &&
                    (C_23 : ((\result == (− 1)) || (\result == 0)))));
{
   (var int32 ___retres);
   { (if (! (r != null)) then
      { (C_1 : (___retres = −1));
         (goto return_label)
      } else ());
      (if ((C_9 : i.int_M) < (C_8 : j.int_M)) then (C_7 : ((C_6 : r.int_M) =
                                                     (C_5 : j.int_M))) else
      (C_4 : ((C_3 : r.int_M) = (C_2 : i.int_M))));
      (C_10 : (___retres = 0));
      (return_label :
      (return ___retres))
   }
}
```

```
48 int32 main(int32 v_a, int32 b)
49 behavior default:
50   assumes true;
51   ensures (C_34 : true);
52 {
53    (var int_P[0] a);
54    (var int_P[0] c);
55    { (C_25 : (a = (C_24 : (new int_P[1]))));
56       (C_27 : ((C_26 : a.int_M) = v_a));
57       (C_29 : (c = (C_28 : (new int_P[1]))));
58       (C_30 : max(c, a, a));
59       { (C_31 : (free(a)));
60          (C_32 : (free(c)));
61          (return (C_33 : c.int_M))
62       }
63    }
64 }
```

Listing A.1: Jessie code of the sample C program

A.3 Why code of the sample C program (abridged)

```
1  parameter main : v_a:int32 −> b:int32 −> { } int32 { true }
2
3  parameter main_requires : v_a:int32 −> b:int32 −> { } int32 { true }
4
5  parameter max :
6   r:int_P pointer −>
7    i:int_P pointer −>
8     j:int_P pointer −>
9      int_P_int_M_r_1:(int_P, int32) memory ref −>
10      int_P_j_3_alloc_table:int_P alloc_table −>
11       int_P_i_2_alloc_table:int_P alloc_table −>
12        int_P_r_1_alloc_table:int_P alloc_table −>
13         int_P_int_M_j_3:(int_P, int32) memory −>
14          int_P_int_M_i_2:(int_P, int32) memory −>
15           { } int32 writes int_P_int_M_r_1
16           { (JC_20:
17             ((JC_18:
18              eq_int(integer_of_int32(select(int_P_int_M_i_2, i@)),
19              integer_of_int32(select(int_P_int_M_i_2@, i@))))
20             and (JC_19:
21                 eq_int(integer_of_int32(select(int_P_int_M_j_3, j@)),
22                 integer_of_int32(select(int_P_int_M_j_3@, j@)))))) }
23
24 parameter max_requires :
25  r:int_P pointer −>
26   i:int_P pointer −>
27    j:int_P pointer −>
28     int_P_int_M_r_1:(int_P, int32) memory ref −>
```

```
29         int_P_j_3_alloc_table:int_P alloc_table −>
30          int_P_i_2_alloc_table:int_P alloc_table −>
31           int_P_r_1_alloc_table:int_P alloc_table −>
32            int_P_int_M_j_3:(int_P, int32) memory −>
33             int_P_int_M_i_2:(int_P, int32) memory −>
34              { (JC_6:
35                ((JC_1: le_int(offset_min(int_P_i_2_alloc_table, i), (0)))
36                and ((JC_2: ge_int(offset_max(int_P_i_2_alloc_table, i), (0)))
37                    and ((JC_3:
38                         le_int(offset_min(int_P_j_3_alloc_table, j), (0)))
39                        and ((JC_4:
40                             ge_int(offset_max(int_P_j_3_alloc_table, j), (0)))
41                            and (JC_5:
42                               ((r = null)
43                               or (le_int(offset_min(int_P_r_1_alloc_table, r),
44                                   (0))
45                                  and ge_int(offset_max(int_P_r_1_alloc_table,
46                                             r),
47                                     (0))))))))))}
48              int32 writes int_P_int_M_r_1
49              { (JC_20:
50                ((JC_18:
51                 eq_int(integer_of_int32(select(int_P_int_M_i_2, i@)),
52                 integer_of_int32(select(int_P_int_M_i_2@, i@))))
53                and (JC_19:
54                    eq_int(integer_of_int32(select(int_P_int_M_j_3, j@)),
55                    integer_of_int32(select(int_P_int_M_j_3@, j@)))))) }
56
57  parameter max_requires_specialized :
58   r:int_P pointer −>
59    i:int_P pointer −>
60     j:int_P pointer −>
61      int_P_int_M_r_1:(int_P, int32) memory ref −>
62       int_P_j_3_alloc_table:int_P alloc_table −>
63        int_P_i_2_alloc_table:int_P alloc_table −>
64         int_P_r_1_alloc_table:int_P alloc_table −>
65          { (JC_6:
66            ((JC_1: le_int(offset_min(int_P_i_2_alloc_table, i), (0)))
67            and ((JC_2: ge_int(offset_max(int_P_i_2_alloc_table, i), (0)))
68                and ((JC_3: le_int(offset_min(int_P_j_3_alloc_table, j), (0)))
69                    and ((JC_4:
70                         ge_int(offset_max(int_P_j_3_alloc_table, j), (0)))
71                        and (JC_5:
72                           ((r = null)
73                           or (le_int(offset_min(int_P_r_1_alloc_table, r),
74                               (0))
75                              and ge_int(offset_max(int_P_r_1_alloc_table, r),
76                                 (0))))))))))}
77          int32 writes int_P_int_M_r_1
```

```
78             { (JC_20:
79               ((JC_18:
80                eq_int(integer_of_int32(select(int_P_int_M_r_1, i@)),
81                integer_of_int32(select(int_P_int_M_r_1@, i@))))
82               and (JC_19:
83                   eq_int(integer_of_int32(select(int_P_int_M_r_1, j@)),
84                   integer_of_int32(select(int_P_int_M_r_1@, j@)))))) }
85
86 parameter max_specialized :
87  r:int_P pointer ->
88   i:int_P pointer ->
89    j:int_P pointer ->
90     int_P_int_M_r_1:(int_P, int32) memory ref ->
91      int_P_j_3_alloc_table:int_P alloc_table ->
92       int_P_i_2_alloc_table:int_P alloc_table ->
93        int_P_r_1_alloc_table:int_P alloc_table ->
94         { } int32 writes int_P_int_M_r_1
95         { (JC_20:
96           ((JC_18:
97            eq_int(integer_of_int32(select(int_P_int_M_r_1, i@)),
98            integer_of_int32(select(int_P_int_M_r_1@, i@))))
99           and (JC_19:
100              eq_int(integer_of_int32(select(int_P_int_M_r_1, j@)),
101              integer_of_int32(select(int_P_int_M_r_1@, j@)))))) }
102
103 parameter safe_int32_of_integer_ :
104  x:int -> { } int32 { eq_int(integer_of_int32(result), x) }
105
106 parameter safe_int8_of_integer_ :
107  x:int -> { } int8 { eq_int(integer_of_int8(result), x) }
108
109 let main_ensures_default =
110  fun (v_a : int32) (b : int32) ->
111   { (JC_32: true) }
112   (init:
113   (let return = ref (any_int32 void) in
114   try
115    begin
116      (let int_P_int_M_a_6 = ref (any_memory void) in
117      (let int_P_a_6_tag_table = ref (any_tag_table void) in
118      (let int_P_a_6_alloc_table = ref (any_alloc_table void) in
119      (let a = ref (any_pointer void) in
120      (let c = (any_int32 void) in
121      (C_23:
122      (C_25:
123      (C_26:
124      (C_27:
125      begin
126        (let jessie_51 =
```

```
127          (a := (C_22:
128                (JC_41:
129                (((alloc_struct_int_P (1)) int_P_a_6_alloc_table)
                       int_P_a_6_tag_table)))) in
130          void);
131         (let jessie_53 = v_a in
132         (let jessie_54 = !a in
133         (((safe_upd_ int_P_int_M_a_6) jessie_54) jessie_53)));
134         (let jessie_60 =
135         (let jessie_59 = !a in
136         (let jessie_58 = !a in
137         (let jessie_57 = !a in
138         (JC_42:
139         (((((((max_specialized jessie_59) jessie_58) jessie_57) int_P_int_M_a_6) !
                int_P_a_6_alloc_table) !int_P_a_6_alloc_table) !
                int_P_a_6_alloc_table))))) in
140         void); ((safe_free_parameter int_P_a_6_alloc_table) !a); (return := c);
141         (raise Return) end))))))))); absurd end with Return −> !return end))
142     { (JC_33: true) }
143
144 let main_safety =
145  fun (v_a : int32) (b : int32) −>
146     { (JC_32: true) }
147     (init:
148     (let return = ref (any_int32 void) in
149     try
150       begin
151         (let int_P_int_M_a_6 = ref (any_memory void) in
152         (let int_P_a_6_tag_table = ref (any_tag_table void) in
153         (let int_P_a_6_alloc_table = ref (any_alloc_table void) in
154         (let a = ref (any_pointer void) in
155         (let c = (any_int32 void) in
156         (C_23:
157         (C_25:
158         (C_26:
159         (C_27:
160         begin
161           (let jessie_39 =
162           (a := (let jessie_38 =
163                 (C_22:
164                 (JC_37:
165                 (((alloc_struct_int_P_requires (1)) int_P_a_6_alloc_table)
                        int_P_a_6_tag_table))) in
166                 (JC_38:
167                 (assert
168                 { ge_int(offset_max(int_P_a_6_alloc_table, jessie_38), (0)) };
169                 jessie_38)))) in void);
170         (let jessie_41 = v_a in
171         (let jessie_42 = !a in
```

```
172          (((safe_upd_ int_P_int_M_a_6) jessie_42) jessie_41)));
173          (let jessie_48 =
174          (let jessie_47 = !a in
175          (let jessie_46 = !a in
176          (let jessie_45 = !a in
177          (JC_40:
178          (assert
179          { (pset_disjoint(pset_singleton(a), pset_singleton(a))
180            and pset_disjoint(pset_singleton(a), pset_singleton(a))) };
181          (JC_39:
182          (((((((max_requires_specialized jessie_47) jessie_46) jessie_45)
                 int_P_int_M_a_6) !int_P_a_6_alloc_table) !int_P_a_6_alloc_table) !
                 int_P_a_6_alloc_table))))))) in
183          void); ((free_parameter int_P_a_6_alloc_table) !a); (return := c);
184          (raise Return) end))))))))); absurd end with Return -> !return end))
185    { true }
186
187  let max_ensures_default =
188   fun (r : int_P pointer) (i : int_P pointer) (j : int_P pointer) (int_P_int_M_r_1 : (
        int_P, int32) memory ref) (int_P_r_1_alloc_table : int_P alloc_table) (
        int_P_i_2_alloc_table : int_P alloc_table) (int_P_j_3_alloc_table : int_P
        alloc_table) (int_P_int_M_i_2 : (int_P, int32) memory) (int_P_int_M_j_3 : (
        int_P, int32) memory) ->
189    { (JC_13:
190      ((JC_8: le_int(offset_min(int_P_i_2_alloc_table, i), (0)))
191      and ((JC_9: ge_int(offset_max(int_P_i_2_alloc_table, i), (0)))
192          and ((JC_10: le_int(offset_min(int_P_j_3_alloc_table, j), (0)))
193              and ((JC_11: ge_int(offset_max(int_P_j_3_alloc_table, j), (0)))
194                  and (JC_12:
195                      ((r = null)
196                      or (le_int(offset_min(int_P_r_1_alloc_table, r), (0))
197                         and ge_int(offset_max(int_P_r_1_alloc_table, r), (0))))))))))
                            }
198    (init:
199    (let return = ref (any_int32 void) in
200    try
201     begin
202       (let ___retres = ref (any_int32 void) in
203       try
204        (let jessie_36 =
205        (C_10:
206        begin
207          (if (not ((safe_neq_pointer r) null))
208          then
209           (let jessie_22 =
210           (C_1:
211           begin
212             (let jessie_21 =
213             (___retres := (safe_int32_of_integer_ (neg_int (1)))) in void);
```

```
214             (raise (Return_label_exc void)) end) in void) else void);
215          (let jessie_33 =
216          (if ((lt_int_ (integer_of_int32 (C_9: ((safe_acc_ int_P_int_M_i_2) i))))
217               (integer_of_int32 (C_8: ((safe_acc_ int_P_int_M_j_3) j))))
218          then
219           (C_7:
220           (let jessie_24 = (C_5: ((safe_acc_ int_P_int_M_j_3) j)) in
221           begin
222             (let jessie_25 = r in
223             (((safe_upd_ int_P_int_M_r_1) jessie_25) jessie_24)); jessie_24 end))
224          else
225           (C_4:
226           (let jessie_29 = (C_2: ((safe_acc_ int_P_int_M_i_2) i)) in
227           begin
228             (let jessie_30 = r in
229             (((safe_upd_ int_P_int_M_r_1) jessie_30) jessie_29)); jessie_29 end))) in
230          void);
231          (let jessie_35 = (___retres := (safe_int32_of_integer_ (0))) in void);
232          (raise (Return_label_exc void)) end) in void) with
233         Return_label_exc jessie_1 ->
234         (return_label: begin (return := !___retres); (raise Return) end) end);
235       absurd end with Return -> !return end))
236     { (JC_17:
237       ((JC_15:
238        eq_int(integer_of_int32(select(int_P_int_M_i_2, i@)),
239        integer_of_int32(select(int_P_int_M_i_2@, i@))))
240       and (JC_16:
241            eq_int(integer_of_int32(select(int_P_int_M_j_3, j@)),
242            integer_of_int32(select(int_P_int_M_j_3@, j@)))))) }
243
244 let max_safety =
245  fun (r : int_P pointer) (i : int_P pointer) (j : int_P pointer) (int_P_int_M_r_1 : (
        int_P, int32) memory ref) (int_P_r_1_alloc_table : int_P alloc_table) (
        int_P_i_2_alloc_table : int_P alloc_table) (int_P_j_3_alloc_table : int_P
        alloc_table) (int_P_int_M_i_2 : (int_P, int32) memory) (int_P_int_M_j_3 : (
        int_P, int32) memory) ->
246     { (JC_13:
247       ((JC_8: le_int(offset_min(int_P_i_2_alloc_table, i), (0)))
248       and ((JC_9: ge_int(offset_max(int_P_i_2_alloc_table, i), (0)))
249            and ((JC_10: le_int(offset_min(int_P_j_3_alloc_table, j), (0)))
250                 and ((JC_11: ge_int(offset_max(int_P_j_3_alloc_table, j), (0)))
251                      and (JC_12:
252                           ((r = null)
253                           or (le_int(offset_min(int_P_r_1_alloc_table, r), (0))
254                              and ge_int(offset_max(int_P_r_1_alloc_table, r), (0))))))))))
                                   }
255     (init:
256     (let return = ref (any_int32 void) in
257     try
```

```
258  begin
259    (let ___retres = ref (any_int32 void) in
260    try
261     (let jessie_18 =
262     (C_10:
263     begin
264       (if (not ((neq_pointer r) null))
265       then
266        (let jessie_4 =
267        (C_1:
268        begin
269          (let jessie_3 =
270          (___retres := (safe_int32_of_integer_ (neg_int (1)))) in void);
271         (raise (Return_label_exc void)) end) in void) else void);
272      (let jessie_15 =
273      (if ((lt_int_ (integer_of_int32 (C_9:
274                                        (JC_23:
275                                        (((acc_ int_P_i_2_alloc_table)
                                              int_P_int_M_i_2) i)))))
276            (integer_of_int32 (C_8:
277                              (JC_24:
278                              (((acc_ int_P_j_3_alloc_table) int_P_int_M_j_3) j))
                                    )))
279      then
280       (C_7:
281       (let jessie_6 =
282       (C_5: (JC_25: (((acc_ int_P_j_3_alloc_table) int_P_int_M_j_3) j))) in
283       begin
284         (let jessie_7 = r in
285         (JC_28:
286         ((((upd_ int_P_r_1_alloc_table) int_P_int_M_r_1) jessie_7) jessie_6)));
287        jessie_6 end))
288      else
289       (C_4:
290       (let jessie_11 =
291       (C_2: (JC_26: (((acc_ int_P_i_2_alloc_table) int_P_int_M_i_2) i))) in
292       begin
293         (let jessie_12 = r in
294         (JC_27:
295         ((((upd_ int_P_r_1_alloc_table) int_P_int_M_r_1) jessie_12) jessie_11)
               ));
296        jessie_11 end))) in void);
297      (let jessie_17 = (___retres := (safe_int32_of_integer_ (0))) in void);
298      (raise (Return_label_exc void)) end) in void) with
299     Return_label_exc jessie_1 ->
300     (return_label: begin (return := !___retres); (raise Return) end) end);
301   absurd end with Return -> !return end)) { true }
```

Listing A.2: Why code of the sample C program (abridged)

A.4 The `DoParse()` function of Jessifier

```
1          public static bool DoParsing()
2          {
3              bool retval = false;
4              // check if we have disjoint contraints
5              if (!Regex.IsMatch(Files.WhyFile, Patterns.Disjoint))
6              {
7                  Errors.ReportError("no_disjoint_info", true);
8              }
9              else
10             {
11                 Report.Print(2, "Some separation analysis assertions are found.");
12             }
13
14             var asserts = Regex.Matches(Files.WhyFile, Patterns.TotalAsserts);
15             Report.Print(2, "{0} asserts found", asserts.Count);
16             foreach (Match mm in asserts)
17             {
18                 Match m = Regex.Match(mm.ToString(), Patterns.Asserts);
19                 Report.Print(1, "At pos {0} : {1} ", m.Index + mm.Index, m.Groups["
                       params"].ToString());
20                 var @params = Regex.Matches(m.ToString(), Patterns.Params);
21                 if (@params.Count > 0)
22                 {
23                     Report.Print(1, "{0} disjoint parameters found", @params.Count);
24                     newsuggest.AppendLine("Some function specification might allow
                           aliases");
25
26                     string func = Regex.Match(mm.ToString(), Patterns.Func).Groups["
                           func"].ToString();
27                     var parmatches = Regex.Matches(mm.ToString(), Patterns.
                           FuncParams);
28                     string[] funcparams = (from Match fm in parmatches
29                                            select fm.Groups["p"].ToString().Replace('!',
                                                  '&')).ToArray<string>();
30
31                     List<Parameter> myparams = new List<Parameter>();
32
33                     foreach (string p in funcparams) myparams.Add(new Parameter {
                           CalledAs = p });
34
35
36                     Report.Print(3, "Separation constraints found for the following
                           invocation:");
37                     string funcall = String.Format("{0}({1});", func, String.Join(",",
                           funcparams));
38                     Report.Print(3, " ==> {0}", funcall);
39
```

```
40                  Report.Print(3, "Looking for function declaration");
41                  string[] funcdeclars = (from Match dec in Regex.Matches(Files.
                        VCCFile, String.Format(Patterns.FuncDeclaration, func) +
                        Patterns.FuncDeclaration2)
42                                  // where Regex.Matches(dec.ToString(),
                                        Patterns.VCCFuncParams).Count ==
                                        funcparams.Length + 1
43                                  select dec.ToString()).ToArray<string>();
44
45
46
47
48                  Report.Print(3, "{0} different declaration(s) found", funcdeclars.
                        Length);
49
50                  foreach (string fun in funcdeclars)
51                  {
52                      string[] funpars = (from Match par in Regex.Matches(Regex.
                            Match(fun, string.Format(Patterns.FuncDeclaration, func)).
                            ToString(), Patterns.VCCFuncParams)
53                                  select par.Groups[1].ToString()).ToArray<
                                        string>();
54                      try
55                      {
56                          for (int i = 0; i < funpars.Length - 1; i++) myparams[i].
                                DeclaredAs = funpars[i + 1];
57                      }
58                      catch (Exception ex)
59                      {
60                          Report.Print(3, "Function declaration deviates from its
                                invocation, skipping declaration...");
61                          continue;
62                      }
63                      Report.Print(2, "Function invocation resolved as:");
64                      Report.Print(2, " {0} <=> {1} ", fun, funcall);
65
66                      newsuggest.AppendLine("Following function call was resolved
                            to have potential aliases:");
67                      newsuggest.AppendLine(funcall);
68
69                      newsuggest.AppendLine("Possible declaration for this call:");
70                      newsuggest.AppendLine(fun);
71
72                      Report.Print(1, "Looking for input parameters");
73
74                      //fetching data
75
76                      foreach (Parameter pr in myparams)
77                      {
```

```
78                  pr.InputParameter = Regex.IsMatch(fun, string.Format(
                        Patterns.VCCInputParams, pr.DeclaredAs));
79                  pr.OutputParamter = Regex.IsMatch(fun, string.Format(
                        Patterns.VCCOutputParams, pr.DeclaredAs));
80              }
81
82              int inputparnum = myparams.Count<Parameter>(a => a.
                    InputParameter);
83
84              Report.Print(2, "{0} input params", inputparnum);
85
86              int outputparnum = myparams.Count<Parameter>(a => a.
                    OutputParamter);
87
88              if (outputparnum == 0)
89              {
90                  Report.Print(3, "No output parameter found, VCC
                        specification is incorrect or incomplete");
91                  return false;
92              }
93
94              // if num of clauses equals num of input params -- one clause
                    for every param
95              if ((@params.Count == inputparnum))
96              {
97                  // if outputparam is only one, we don't have to look for an
                        appopriate one
98                  if (outputparnum == 1)
99                  {
100                     StringBuilder ct = new StringBuilder();
101                     //generating preconditions
102                     Report.Print(3, "Generating pre-conditions...");
103                     newsuggest.AppendLine("These pre-conditions were
                            automatically generated:");
104                     string outputparam = myparams.Where<Parameter>(
                            a => a.OutputParamter).First<Parameter>().
                            DeclaredAs;
105                     foreach (Parameter par in myparams.Where<
                            Parameter>(a => a.InputParameter))
106                     {
107                         ct.AppendLine(String.Format("ensures (*{0} !=
                                *{1}) // <= autogenerated", par.DeclaredAs,
                                outputparam));
108                         newsuggest.AppendLine(String.Format("ensures
                                (*{0} != *{1}) // <= autogenerated", par.
                                DeclaredAs, outputparam));
109                     }
110                     contracts.Append(ct.ToString());
111
```

```
112                             //modifying VCC
113                             Report.Print(3, "Modifying VCC file...");
114                             Match writecontracts = Regex.Match(newvcc.ToString
                                    (), string.Format(Patterns.FuncDeclaration, func));
115                             int insertpos = writecontracts.Index + writecontracts.
                                    Length + 1;
116                             newvcc = new StringBuilder(newvcc.ToString().
                                    Substring(0, insertpos) + ct.ToString() + newvcc.
                                    ToString().Substring(insertpos, newvcc.ToString().
                                    Length − insertpos));
117                             retval = true;
118                         }
119                         else
120                         {
121                             Errors.PrintError("suggest", true);
122                         }
123                     }
124                     else
125                     {
126                         Errors.PrintError("suggest", true);
127                     }
128
129                 }
130                 Report.Print(3, "No more function declarations found");
131                 return retval;
132
133             }
134             else
135             {
136                 Report.Print(1, "No disjoint parameters found, skipping .....");
137                 newsuggest.AppendLine("No possible aliases were detected in code")
                        ;
138                 return false;
139             }
140
141         }
142
143         return true;
144     }
```

Listing A.3: Source code for the DoParse() function of Jessifier

A.5 Regular expressions used for code parsing

```
1   public struct Patterns
2   {
3       public const string Disjoint = @"pset_disjoint\(pset_singleton\((?<par1>.)\)
            \,\W∗pset_singleton\((?<par2>.)\)\)";
4       public const string DisjointGroups = @"(?:pset_disjoint\(pset_singleton\((?<p1
            >.)\)\,\W∗pset_singleton\((?<p2>.)\)\)(?:\W∗(?<op>and)\W∗)?)(?:
```

```
            pset_disjoint\(pset_singleton\((?<p3>.)\)\,\W*pset_singleton\((?<p4>.)
            \)\))?";
5       public const string TotalAsserts = @"\(let\W+jessie_(?:\d+)\W+=\W+ (?:\(
            let\W+jessie_(?:\d+)\W+(\w)\Win[^\(]*)+\(JC_(?:\d+):\W*\(assert\W
            *\{([^\}]+)\}([^_][^r][^e])*";
6       public const string Asserts = @"assert\W*\{(?<params>[^\}]+)\}";
7       public const string Params = @"(?:pset_disjoint\(pset_singleton\((?<p1>.)\)
            \,\W*pset_singleton\((?<p2>.)\)\))";
8       public const string Func = @"(?<func>\w+)_requires";
9       public const string FuncParams = @"\(let\W+jessie_(?:\d+)\W+=\W+(?<p
            >[!]\w)+\W+in";
10      public const string FuncDeclaration = @"(?:\w+)\W+{0}\((\W?\w+\W?[*]?\
            W?\w+[,]?)*\W?\)[^;]";
11      public const string FuncDeclaration2 = @"([^\{]*)";
12      public const string VCCFuncParams = @"\W*\w+\W?[*]*\W?(\w+)\W?[,]?";
13      public const string VCCInputParams = @"\W*{0}\W*==\W*old\(\W*{0}\W
            *\)";
14      public const string VCCOutputParams = @"\W*writes\W*\(\W*({0})\W*\)
            ";
15
16  }
```

Listing A.4: Regular expressions used for code parsing

Bibliography

[Appel 07] A. Appel & S. Blazy. *Separation logic for small-step C minor.* In 20th International Conerence On Theorem Proving In Higher-Oder Logics (TPHOLS), pages 5–21. Springer, 2007.

[Ball 04] T. Ball, B. Cook, V. Levin & S.K. Rajamani. *SLAM and Static Driver Verifier: Technology transfer of formal methods inside Microsoft.* In Integrated Formal Methods, pages 1–20. Springer, 2004.

[Barras 08] B. Barras, S. Boutin, C. Cornes, J. Courant, Y. Coscoy, D. Delahaye, D. de Rauglaudre, J.C. Filliâtre, E. Giménez, H. Herbelin*et al. The Coq proof assistant reference manual: Version 8.* In INRIA, 2008.

[Barthe 06] G. Barthe, J. Forest, D. Pichardie & V. Rusu. *Defining and reasoning about recursive functions: a practical tool for the Coq proof assistant.* In Functional and Logic Programming (FLOPS'06), LNCS 3945, pages 114–129. Springer, 2006.

[Baumann 09a] C. Baumann, B. Beckert, H. Blasum & T. Bormer. *Better Avionics Software Reliability by Code Verification.* In M. Sturm, editeur, Embedded World 2009 Proceedings & Conference Materials, Nuremberg, Germany, 2009. WEKA FACHMEDIEN GmbH.

[Baumann 09b] C. Baumann, B. Beckert, H. Blasum & T. Bormer. *Formal Verification of a Microkernel Used in Dependable Software Systems.* In B. Buth, G. Rabe & T. Seyfarth, editeurs, Computer Safety, Reliability, and Security (Safecomp 2009), LNCS 5775, pages 187–200. Springer Berlin / Heidelberg, 2009.

[Baumann 10] C. Baumann, B. Beckert, H. Blasum & T. Bormer. *Ingredients of Operating System Correctness.* In M. Sturm, editeur, Embedded World 2010 Proceedings & Conference Materials, Nuremberg, Germany, 2010. WEKA FACHMEDIEN GmbH.

[Berdine 05] J. Berdine, C. Calcagno & P. O'Hearn. *Smallfoot: Modular Automatic Assertion Checking with Separation Logic.* In F.S. de Boer, M.M. Bonsangue, Graf S. & W.P. de Roever, editeurs, Formal Methods for Components and Objects (FMCO), volume 4111, pages 115–137, Berlin, 2005. Springer Verlag.

[Böhme 08] S. Böhme, K. Leino & B. Wolff. *HOL-Boogie: an interactive prover for the Boogie program-verifier.* In O.A. Mohamed, C. Muñoz & S. Tahar, editeurs, Theorem Proving in Higher Order Logics, volume 5170 of *Lecture Notes in Computer Science*, pages 150–166. Springer, 2008.

[Böhme 10] S. Böhme, M. Moskal, W. Schulte & B. Wolff. *HOL-Boogie: An Interactive Prover-Backend for the Verifying C Compiler.* Journal of Automated Reasoning (JAR), vol. 44, no. 1-2, pages 111–144, 2010.

[Botincan 09] M. Botincan, M. Parkinson & W. Schulte. *Separation logic verification of C programs with an SMT solver.* Electronic Notes in Theoretical Computer Science, vol. 254, pages 5–23, 2009.

[Burghardt 11] J. Burghardt, J. Gerlach, L. Gu, K. Hartig, H. Pohl, J. Soto & K. Völlinger. ACSL by example: Towards a verified c standard library. Frauenhofer First, 2011.

[Cohen 09a] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte & S. Tobies. *VCC: A practical system for verifying concurrent C.* In 22th International Conerence On Theorem Proving In Higher-Oder Logics (TPHOLS), pages 23–42, 2009.

[Cohen 09b] E. Cohen, M. Moskal, W. Schulte & S. Tobies. *A Practical Verification Methodology for Concurrent Programs.* Rapport technique MSR-TR-2009-15, Microsoft Research, February 2009.

[Cohen 09c] E. Cohen, M. Moskal, S. Tobies & W. Schulte. *A Precise Yet Efficient Memory Model for C.* In 4th International Workshop on Systems Software Verification (SSV 2009), volume 254 of *Electronic Notes in Theoretical Computer Science*, pages 85–103. Elsevier Science B.V., 2009.

[Cohen 10a] E. Cohen, E. Alkassar, V. Boyarinov, M. Dahlweid, U. Degenbaev, M. Hillebrand, B. Langenstein, D. Leinenbach, M. Moskal, S. Obua, W. Paul, H. Pentchev, E. Petrova, T. Santen, N. Schirmer, S. Schmaltz, A. Shadrin, S. Tobies, A. Tsyban & S. Tverdyshev. *Invariants, Modularity, and Rights.* In A. Pnueli, I. Virbitskaite & A. Voronkov, editeurs, Perspectives of Systems Informatics, volume 5947 of *Lecture Notes in Computer Science*, pages 43–55. Springer Berlin / Heidelberg, 2010.

[Cohen 10b] E. Cohen, W. Moskal M. Schulte & S Tobies. *Local Verification of Global Invariants in Concurrent Programs.* In B. Cook, P. Jackson & Touili, editeurs, Computer Aided Verification (CAV 2010), 2010.

[Cohen 10c] E. Cohen, S. Tobies, M. Moskal & W. Schulte. *Verifying Concurrent C Programs with VCC*, 2010.

[Dahlweid 09] M. Dahlweid, M. Moskal, T. Santen, S. Tobies & W. Schulte, editeurs. VCC: Contract-based modular verification of concurrent C, 2009.

[Detlefs 05] D. Detlefs, G. Nelson & J.B. Saxe. *Simplify: A theorem prover for program checking.* Journal of the ACM (JACM), vol. 52, no. 3, pages 365–473, 2005.

[Dijkstra 72] E.W. Dijkstra. *The humble programmer.* Communications of the ACM, vol. 15, no. 10, pages 859–866, 1972.

[Dijkstra 97] E.W. Dijkstra. *Answers to questions from students of software engineering*, 1997.

[Distefano 08] D. Distefano, J. Parkinson & J. Matthew. *jStar: Towards practical verification for Java.* ACM SIGPLAN Notices, vol. 43, no. 10, pages 213–226, 2008.

[Henzinger 03] T. Henzinger, R. Jhala, R. Majumdar & G. Sutre. *Software verification with BLAST.* In Proceedings of the 10th international conference on Model checking software, SPIN'03, pages 235–239, Berlin, Heidelberg, 2003. Springer-Verlag.

[Hubert 07] T. Hubert & C. Marché. *Separation analysis for deductive verification*. In Heap Analysis and Verification (HAV'07), pages 81–93, 2007.

[Hubert 08] T. Hubert & C. Marché. *Separation Analysis for Weakest Precondition-based Verification*, 2008.

[Jacobs 11] B. Jacobs & F. Piessens. *Expressive modular fine-grained concurrency specification*. In Proceedings of the 38th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '11, pages 271–282. ACM, 2011.

[Janota 10] M. Janota. *SAT Solving in Interactive Configuration*. PhD thesis, University College Dublin, 2010.

[Jetley 09] R. Jetley & B. Chelf. *Diagnosing Medical Device Software Defects using Static Analysis*. Medical Device and Diagnostic Industry, vol. 31:5, pages 72–83, 2009.

[Leinenbach 09] D. Leinenbach & T. Santen. *Verifying the Microsoft Hyper-V Hypervisor with VCC*. In 16th International Symposium on Formal Methods (FM 2009), LNCS 5850, pages 806–809, Eindhoven, the Netherlands, 2009. Springer.

[Marché 10] C. Marché & Y. Moy. *Jessie Plugin Tutorial*, 2010.

[Maus 08] S. Maus, M. Moskal & W. Schulte. *Vx86: x86 assembler simulated in C powered by automated theorem proving*. Algebraic Methodology and Software Technology, pages 284–298, 2008.

[Moskal 09] M. Moskal. *Satisfiability Modulo Software*. PhD thesis, University of Wroclaw, 2009.

[Moy 09] Y. Moy. *Automatic Modular Static Safety Checking for C Programs*. PhD thesis, Université de Paris-Sud, 2009.

[Nguyen 07] H. Nguyen, C. David, S. Qin & W.N. Chin. *Automated verification of shape and size properties via separation logic*. In Verification, Model Checking, and Abstract Interpretation, pages 251–266. Springer, 2007.

[Nipkow 11] T. Nipkow, L. Paulson & M. Wenzel. Isabelle HOL: A proof assistant for higher-order logic. Springer Verlag, 2011.

[Pierce 96] P. Pierce. *Software verification and validation*. In Northcon/96, pages 265–268, 1996.

[Wand 77] M. Wand. *A characterization of weakest preconditions*. Journal of Computer and System Sciences, vol. 15, no. 2, pages 209–212, 1977.