# Verification of Cyber Physical Systems

Dilip Venkateswaran Murali

Thesis submitted to the faculty of the Virginia Polytechnic Institute and State University in partial fulfillment of the requirements for the degree of

Master of Science

In

Computer Engineering

Michael S. Hsiao, Chair

A. Lynn Abbott

Sandeep K. Shukla

30 August, 2013

Blacksburg, Virginia

Keywords: Invariants detection, Symbolic Execution, KLEE, Cloud9, VCC

# Verification of Cyber Physical Systems

Dilip Venkateswaran Murali

# (ABSTRACT)

Due to the increasing complexity of today's cyber-physical systems, defects become inevitable and harder to detect. The complexity of such software is generally huge, with millions of lines of code. The impact of failure of such systems could be hazardous. The reliability of the system depends on the effectiveness and rigor of the testing procedures. Verification of the software behind such cyber-physical systems is required to ensure stability and reliability before the systems are deployed in field. We have investigated the verification of the software for Autonomous Underwater Vehicles (AUVs) to ensure safety of the system at any given time in the field. To accomplish this, we identified useful invariants that would aid as monitors in detecting abnormal behavior of the software. Potential invariants were extracted which had to be validated. The investigation attempts to uncover the possibility of performing this method on existing Software verification platforms. This was accomplished on Cloud9, which is built on KLEE and using the Microsoft's VCC tool. Experimental results show that this method of extracting invariants can help in identifying new invariants using these two tools and the invariants identified can be used to monitor the behavior of the autonomous vehicles to detect abnormality and failures in the system much earlier thereby improving the reliability of the system. Recommendations for improving software quality were provided. The work also explored safety measures and standards on software for safety critical systems and Autonomous vehicles. Metrics for measuring software complexity and quality along with the requirements to certify AUV software were also presented. The study helps in understanding verification issues, guidelines and certification requirements.

# Acknowledgements

Firstly, I extend my sincere thanks to, Dr. Michael S. Hsiao, my research advisor, for giving me an opportunity to work in the PROACTIVE (Proactive Research On Advanced Computer-aided TestIng, VErification, and power-management techniques) group, and keeping confidence in me throughout my engagement with the group. His exceptional teaching in the courses, Testing of Digital Systems and Electronic Design Automation, inspired me to join his PROACTIVE research group. His objective approach to work has served as a great motivation to accomplish higher goals, and his continued guidance and feedback has helped me improve on many fronts.

Thanks to Dr. A. Lynn Abbott and Dr. Sandeep K.Shukla, for serving on my thesis committee.

I would like to thank the faculty members of the project – Dr. Michael S. Hsiao, Dr. Craig A. Woolsey, and Dr. Daniel J. Stilwell for providing an interesting platform for brainstorming and discussion of ideas while working as a team.

I would like to express my sincere gratitude to my parents N.V.Murali and Latha Murali, my sister Divya Shantha.M, my extended family and friends for their love, encouragement, and support at every step in my life.

Finally, I would like to thank my lab-mates for providing me an excellent environment within the lab and for the great discussions on both technical and non-technical topics.

Dilip Venkateswaran Murali

August 2013

# Table of Contents

# List of Figures

# List of Tables

# 1. Introduction

Software plays a fundamental role in our society, bringing enormous benefits to various fields. Software systems have become an inevitable part of today's world. They are pervasive and back diverse applications ranging from defense missile systems and airplane navigation systems to common electronic appliances like mobile phones and remote controlled toys. Such software applications fuel complex business processes, next-generation interactive TVs, communication infrastructures and a myriad of internet services. And such software-based systems are rapidly replacing older technologies in numerous applications.

Software systems have become a necessity to every industry. With the rise in demand and the diversity in the application areas, the functionality required from these systems has also increased. Consequently, the software becomes more complex and gets bigger in terms of size. Due to the increasing complexity of today's systems, defects become inevitable and harder to detect.

The challenge in developing such systems lies in the ability to deliver quality code with fewer defects. Although these systems are tested before release there are still bugs undetected in the system which requires bug fixes in the next release. Since many of our current systems are highly centralized and tightly coupled, they are also susceptible to massive and coordinated failure.

Defects undetected after the product has shipped can be expensive to address and fix. The cost for a detected fault in the system increases if it is found later in the software development life cycle, or even in the field. The cost to detect and fix a fault is the highest when detected in the field and the least when it is found in the design phase [1]. Hence, the reliability and integrity of the software becomes highly important both for the industry in terms of cost associated with the failure and for the end user.

According to Jeff Payne, president and Chief Executive Officer of code-assessment services provider Cigital, software failure can be attributed to three reasons:

"First, software is probably the most complex thing we try to build today". Second, "the nature of software is such that no foolproof set of rules can be created that will absolutely eliminate bugs". The third reason is "the fact that developers and people who build software just do a very poor job of testing, validating, and building what they're doing" [2].

As discussed earlier, software systems find various applications in diverse fields. They form the backbone of Cyber Physical Systems (CPS). While it is common to find them in manned vehicles, they are far more critical in unmanned vehicles. With the technology trend moving towards more autonomy in our lives, such as self-monitoring systems and autonomous vehicles, where the intervention of humans is bare minimum, the software needs to have zero or very low defect densities. Any undetected defect must be guarded with some defect-tolerance mechanisms. The complexity of such software is generally huge, with millions of lines of code.

For example, the avionics system in the U.S. Air Force frontline jet fighter, F-22 Raptor, consists of about 1.7 million lines of software code. The F-35 Joint Strike Fighter contains about 5.7 million lines of code to operate its onboard systems. Boeing's new 787 Dreamliner has about 6.5 million lines of software code to operate its avionics and onboard support systems [3].

The following are a few facts and figures from NASA [4]:

- 3.3 million Lines of software code on the ground support 1.8 million lines of flight software code.
- In the International Space Station's U.S. segment alone, 1.5 million lines of flight software code run on 44 computers communicating via 100 data networks transferring 400,000 signals (e.g. pressure or temperature measurements, valve positions, etc.).

The world has seen several bugs in unmanned vehicular systems; some of those failures have been catastrophic. For example, the launch of the maiden flight Ariane 5 rocket, Flight 501, that took place in 1996 ended in a catastrophic failure 40 seconds after the flight initiation sequence, which incurred a direct cost of approximately $370 million. Reports from the Inquiry Board Report (IBR) identified the proximate cause of the disaster as a software failure [5].

Inadequate protection from integer overflow in the software resulted in the failure of the rocket to achieve the orbit. The Ariane 5 had a greater horizontal acceleration which caused the computers in both the back-up and primary platforms to crash and emit diagnostic data which was misinterpreted by the autopilot as spurious position. The acceleration caused a data conversion from a 64-bit floating point number to a 16-bit signed integer value to overflow and cause a hardware exception. The exception halted the reference platforms, resulting in the destruction of the flight. This bug could have been eliminated by the use of appropriate assertions during the development phase. The disaster was a clear result of inadequate validation and verification, testing and review [6]. This case study highlights the impact of a catastrophe of a failure of such systems and the cost effects of failure and emphasizes the importance of verification.

According to a study commissioned by the National Institute of Standards and Technology (NIST), software bugs cost the U.S economy about $59.5 billion annually (about 0.6 percent of the GDP). It was also estimated that more than one third of these costs, about $22.2 billion could be eliminated by improving testing infrastructure that allows early detection and removal of the defects [7].

From the above examples, it is clear that the impact of failure of such systems could be hazardous. In addition, the vehicle should be able to function safely without human inspection. So, the system should automatically be able to avoid dangerous and unsafe situations. Safety is guaranteed by the properties of the system. Software plays a very critical role in detecting the defects in the system, which may also include malfunctioning of the hardware. This leads to the need for the software to be tested extensively and the properties verified.

The reliability of the system depends on the effectiveness and rigor of the testing procedures. For a small piece of software, it is sometimes possible to achieve 100% test coverage with manual test case generation. Manual test cases for large software are typically not exhaustive and both the software and the manual test cases developed are prone to human error. But, even with automatic test coverage tools or test case generators, it is not possible to achieve 100% test coverage for larger programs because of the number of permutations of the states in the program execution and the time taken to exercise all the paths in the code is high. Thus, even software, which has been tested and verified, cannot be assured to be completely error free, since the spec

is often unavailable or incomplete. Also, in most cases testing on the field is very expensive. Therefore, it is desirable that the code is either tested using a simulator or the code is verified by formal methods.

Verification of the software behind such cyber-physical systems is required to ensure safe working of the system before they are deployed in field. To enhance safety in such systems, early detection of errors in the field becomes mandatory to avoid catastrophic failures and result in saving huge amount of resources since in-field testing could be expensive.

## 1.1 Contributions

The objective of this thesis is to investigate the existence of any invariants in a given pseudo-code for Autonomous Underwater Vehicles (AUVs) that can be used to monitor the system to ensure safety of the system at any given time in the field. This investigation also attempts to uncover the possibility of performing this method on existing software verification platforms. It explains the methodical approach that should be followed to verify the code using these specific tools for the AUV. The feasibility study was performed on two tools. Another parallel objective is to explore safety measures and existing standards on software for safety critical systems and autonomous vehicles. These measures and standards aid in comprehension of the requirements imposed on the design and testing of the software and the number of agencies/bodies that approve the software before it is deployed in the field. Parameters and metrics for Software quality and complexity were studied to help the AUV software team to calculate the metrics and to calculate the complexity to define the software quality. The requirements to certify software are also studied to understand verification issues and guidelines, certification requirements for software for AUVs. Finally recommendations are provided to the AUV team that can be incorporated in their software to improve the quality of the entire system.

The approach uses invariant extraction techniques along with symbolic and concolic techniques in conjunction with constraint solvers to produce concrete inputs that test each path in the program under test. Parameters of interest were identified from the given pseudo-code and the code implemented was simulated to obtain a range of values for each of the parameters of interest in a multi-threaded environment. From the simulated values, *potential invariants* were learned and extracted which were verified to be true or false. To verify these invariants, two tools were used:

1. Cloud9
2. VCC

With cloud9, the code had to be modified and run with these potential invariants in a separate monitor thread. The extracted invariants had to exercise different values which satisfy certain known conditions. The invariants were validated with the use of assertions. This strategy exploits capabilities of the symbolic execution engine KLEE, on which cloud9 is built. KLEE attempts to hit every line of the executable code in the program and to detect if there exists an input value that can cause error, at every critical operation. By running programs symbolically, constraints that exactly describe the set of values possible on a given path were generated. When an error is detected or when a path reaches an exit call, KLEE solves the current path's constraints to produce a corresponding test case.

The second tool used was VCC: A Verifier for Concurrent C. The given pseudo-code was converted to C keeping intact the functionality of the modules. The C program was then annotated by specifying pre and post conditions for the functions. This aided in verifying the functions individually for any bugs. By adding other constraints for the various variables, errors caused by overflows could be detected. A new monitor function was written, which included all the parameters of interest from all other functions in the code. Constraints for each parameter which also included the ranges and mathematical relationship between various physical parameters were added as pre-condition, and potential invariants were added as post condition. VCC tries to assign values for each of the parameters, satisfying the behavior specified as annotations, until it tries to find a value that satisfies the pre-condition and ensuring the post-condition. With this technique new invariants were identified which could be used in monitors for safety.

These verified invariants can be added to a monitor thread in the actual system to catch abnormal behavior and abort the system much earlier or before any catastrophic failure can occur. These methods ensure safety and correct functioning of the system against a set of properties that the system must abide by, improves the reliability, and reduces the cost of in-field testing.

## 1.2 Thesis organization

The rest of this thesis is organized as follows:

Chapter 2 presents the background. This section talks about the motivation for the thesis, giving a background of Autonomous Underwater Vehicles (AUVs) and Cyber-Physical Systems. The initiatives on AUV, software metrics and parameters for software certifications and the software certification policies are highlighted. The workings of the tools that have been used to verify the code are explained subsequently.

Chapter 3 briefly explains the pseudo-code provided that explains the functionality of software for AUV, the reason for choosing Cloud9 and explains in detail the sequential methodology followed from writing the software till verifying the code on cloud9.

Chapter 4 explains the detailed procedures for annotating the code and verifying the code using the tool VCC.

Chapter 5 concludes the thesis work, makes recommendations for writing better software and provides the scope and directions for future work.

# 2. Background

## 2.1 Motivation

Autonomous underwater vehicles (AUVs) are untethered mobile instrumentation platforms which have actuators, sensors, and on-board intelligence for their respective application [8] [9].

Unmanned operations without a tether offer great opportunities but pose difficult technical challenges. Elimination of the tether not only frees the vehicle from the surface vessel but also eliminates the large and costly handling gear which a tether entails. Despite the benefits, through-water communication is extremely difficult compared with the above-surface communication. In addition, the vehicle should be able to function safely and productively with little or no human intervention. Thus, the system should be able to automatically avoid dangerous and unsafe situations. Safety is guaranteed by the properties and correctness imposed on the operation of the entire system including both hardware and software components. Reliability of the system is dependent on the correctness of the data provided by sensors and the system's ability to catch hardware malfunction and software bugs. Hence the underlying hardware and software need to be reliable. Software is the primary glue that ties everything together in the system, including the critical component responsible for the detection of hardware/sensor malfunction. This brings in the need for the software to be tested extensively and the properties verified.

Reliability of the software depends on the effectiveness of the test cases and the comprehensiveness of the test inputs. Note that comprehensiveness may not imply explicit exhaustiveness of test cases, if one can demonstrate implicit exhaustiveness. In a system like the AUV where there are multiple processes and threads running in parallel, the behavior of the system could be very unpredictable and the possibilities of failures like Physical failures (caused

due to damage of components), missing Synchronization points, Concurrency bugs or Starvation of a task are even more.

In-field testing of the code may be expensive, including the loss of the AUV during testing. Thus, if we could generate an environment that resembles the actual deployment of the AUV to "simulate" the code, this could reduce cost, the amount of error, bugs, and catastrophic losses before testing it on the field. In most cases, the simulation environments do not fully replicate the actual behavior of the system in field. However, this may work to our advantage, if the inaccuracy offers a level of abstraction (over-approximation) which the verification proof can use.

In short, verification could be applied to improve the reliability of the software, reduce the development risk and reduce the cost of the software product. With the software specifications, verification could be a powerful method for eliminating software errors. This could involve a number of means like mathematical proof for absence of errors in implementations relative to specifications or a formal verification or Model Checker.

The sections below explain in brief about Cyber Physical Systems and Autonomous Underwater Vehicles which is the area that is verified. It also explains about invariants and various other software metrics that are used before explaining about the background work in these areas.

## 2.2 Cyber Physical Systems (CPS)

Cyber Physical Systems (CPSs) are systems in which physical processes are tightly coupled with computation. The computers interact with the physical surroundings using a wide range of

sensors and actuators. They have a feedback loop mechanism to facilitate the interaction. This enables the systems to monitor and control the physical processes and vice versa.

Applications for cyber-physical systems can be found in health care (assisted living, bionics, wearable devices, etc.), transportation and automotive networks, aerospace and avionics, automated manufacturing, blackout-free electricity generation and distribution, optimization of energy consumption in buildings and vehicles, critical infrastructure monitoring, disaster response, efficient agriculture, environmental science, and personal fitness [10].

Autonomous vehicles could also be used in the military to improve its effectiveness in disaster recovery techniques.

Sensing and manipulation of the physical world occurs locally, while monitor and control are enabled safely, globally, securely, reliably and in real-time across a virtual network. This capability is referred to as "Globally Virtual, Locally Physical". Realizing the ample potential of CPS, major investments are being made worldwide to develop the technology.

Cyber-physical systems in most cases are concurrent processes. Coupling the physical processes, which are concurrent in nature, with the computing world, requires the computing environment to be concurrent as well. The focus to make the computing close to parallel dynamics of the physical process makes the complete system fairly complex. Due to their interaction with physical environments which fluctuate very often, these systems are often exposed to highly dynamic set of inputs. They have to be very adaptable to unfavorable conditions, tolerant to subsystem failures and self-diagnostic. Hence, these systems require higher reliability, predictability and robustness to tackle the unpredictability of its surroundings.

With the nondeterministic nature of multi-threaded environment, the burden of replicating the physical world by adding deterministic constraints on the software falls on the programmer [11] [12].

Therefore, the system's efficiency and performance is dependent on the programmer's talent. Nevertheless, irrespective of the programmer's skills, the software is susceptible to human errors. The same is true for the lack of completeness in manually generated test cases. There is a need to bridge the gap between formal methods of verification and testing. Compositional

verification and testing methods are necessary to explore the heterogeneity of the CPS models. Verification and validation is necessary to ensure deterministic, predictable behavior of the system and also be a part of certification regimes for such software [13].

With recent advancements in the field of automated verification, there are formal verification methods, emulation and simulation techniques, certification methods, software engineering processes, design patterns, and software component technologies that can be utilized to its full potential to improve the defect densities. Model-based design where "programs" are replaced by "models" that represent system behaviors of interest can be used as the starting point from which Software can be synthesized [14].

## 2.3 Autonomous Underwater Vehicle (AUV)

Autonomous Underwater Vehicle is one of the applications of Cyber Physical Systems. Autonomous Underwater Vehicles (AUVs) are programmable, robotic vehicles. They have the ability to drift, drive, or glide through the ocean without real-time human intervention. To exercise a certain level of control, AUVs can communicate with human operators either periodically or continuously through satellite signals or underwater acoustic beacons [15].

The benefits of AUVs are multi-fold. An example is that researchers and scientists can conduct experiments from a surface ship while the AUV is off collecting data elsewhere on the surface or in the deep ocean. Some AUVs can be made intelligent enough to change their mission profile based on environmental data that they receive through sensors while under way.

Furthermore, AUVs demonstrate the ease of access in harsh environments like under ice or deepest oceans. This facilitates a high level of persistence for applications that require long-term monitoring. For example, they can be used for survey missions under the water like detecting

and mapping submerged wrecks or other obstructions that make it difficult for the commercial or recreation vessels to navigate.

They are more efficient in collecting data with much higher quality and efficiency than towed systems. They have better maneuverability and navigability than vehicles with tether that allows them to run accurate track lines and follow bottom contours. This enables them to perform very tight turns which increase the ratio of survey time to the time required to position the system for the next track line which enables faster data collection.

The following are some applications where AUVs are employed [16]:

- Commercial Applications
    - In-shore survey applications, such as mapping the seafloor to support maintenance and dredging operations.
    - Off-shore survey applications such as mapping and development of offshore resources such as minerals, oil, and gas.
    - Search and salvage operations to find and recover items of high-value, or detect hazards to navigation, items that can cause environmental damage or items of historical interest, or items that provide forensic data.
- Scientific Applications
    - Oceanography
    - Archaeology and Exploration
    - Environmental Protection and Monitoring
    - Scientific Research
- Defense Applications
    - Port and Harbor Security
    - Ship Hull and Infrastructure Inspection
    - Mine Countermeasures (MCM)
    - Unexploded Ordnance (UXO)
    - Rapid Environmental Assessment (REA)
    - Anti-Submarine Warfare (ASW)
    - Intelligence, Surveillance and Reconnaissance (ISR)

## 2.4 Invariants

Invariants are properties of a system that do not change even when different inputs are applied to them. In mathematics, an invariant is a property of a class of mathematical objects that remains unchanged when transformations of a certain type are applied to the objects. The particular class of objects and type of transformations are usually indicated by the context in which the term is used [17].

An invariant in computer science is a condition that can be relied upon to be true during execution of a program. They are a useful mechanism to ensure correctness of a computer program. When qualifiers are added as prefixes to "invariant", it can yield extended notions of invariants. For example, "loop invariants" are used as conditions that are required to be true at the start and end of every execution of a loop. However, these invariants may not be true elsewhere in the program [18].

For example, consider the following function which divides 'x' by 'd'.

```
void divide(unsigned x, unsigned d, unsigned *q, unsigned *r)
{
        unsigned lq = 0;
        unsigned lr = x;
        while(lr >= d)
        {
                lq++;
                lr -= d;
        }
        *q = lq;
        *r = lr;
}
```

Here an invariant is (x = d*lq + lr). This property of division should always hold at the end of the loop.

Invariants in a program find their uses in many verification tools. Formal methods of verification rely on invariants. Compiler optimization is another area that uses invariants. Invariants can help

avoid human errors to an extent by protecting against inadvertent changes that violate assumptions upon which the program's correct behavior depends [19].

The theory of optimizing compilers, the methodology of design by contract, and formal methods for determining program correctness, all rely heavily on invariants in computer programs. Invariants are usually implicit in program code. Assertions are commonly used by programmers to make some invariants explicit. However, there are some Object Oriented programming languages that have special syntax for specifying class invariants.

Invariants aid in better software delivery at almost every aspect of the Software Development Life Cycle from design, coding, verification, testing and maintenance. A comprehensive understanding of the invariants in the code enhances the programmer’s understanding of the algorithm, program operation and hence in debugging. Despite their advantages, invariants usually need to be identified and discovered. Hence identifying invariants becomes useful to prevent bugs in the software and also to prevent any hazardous events on failure of some component. It also provides additional information to the tools that are used for verification and testing [20].

The sections below highlight the Initiatives and existing standards on unmanned vehicles and certifications for safety critical avionics systems.

## 2.5 Initiatives on unmanned vehicles

There has been several Government and private organizations working together to bring AUV flight operations in line with manned operations.

The following lists some of the Government/Industry initiatives and Associations/Standards towards unmanned vehicles in U.S and other parts of the world [21].

### 2.5.1 Government/industry initiatives

- The Technical Analysis and Applications Center (TAAC)
- Joint Planning and Development Office (JPDO) UAV National Task Force (UNTF).
- UAV National Industry Team (UNITE)
- Access 5
- JAA/Eurocontrol UAV Task Force
- UAV Thematic Network (UAVNET)
- Civil UAV Applications & Economic effectiveness of potential CONfiguration solutions (CAPECON)
- UAV Safety Issues for Civil Operation (USICO)
- UAV's Concerted Actions for Regulations (UCARE)
- Euro UAV Industry Consultative Body (ICB)

### 2.5.2 Associations and standards organizations

Associations have played a key role in bringing various industry and government efforts together to support the creation and expansion of a civil/commercial UAV market.

- The Association for Unmanned Vehicle Systems International (AUVSI)
- American Society for Testing and Materials (ASTM) UAV Committee
- Unmanned Vehicle Systems (UVS) International
- Unmanned Aerial Vehicle Systems (UAVS) Association
- American Institute of Aeronautics and Astronautics (AIAA) Unmanned Systems Program Committee (USPC)
- RTCA, Inc. (formerly Requirements and Technical Concepts for Aviation; and formerly Radio Technical Commission for Aeronautics) (RTCA) Special Committee 203 (SC-203) for UAV Standards Development.

## 2.6 DO-178B overview

This section provides an overview of standards available in the market that ensure correctness of the software for complex systems and the levels of safety associated based on criticality of the software. This section covers DO-178B, a standard for safety critical avionics systems. The standards for UAVs are quite similar in most aspects because of the similarity of the type of system.

DO-178B/ED-12B provides guidance on designing, specifying, developing, testing, and deploying software in safety-critical avionics systems. DO-178B guidelines determine that the software aspects of airborne systems and equipment comply with FAA airworthiness requirements [22].

The avionics industry requires that safety critical software should be created in accordance with the certification authority guidelines before it could be used on any commercial airliner. DO-178B is the current version of DO178, described in the Software Considerations in Airborne Systems and Equipment Certification RTCA/DO-178B Document. DO-178B guidelines are used both by the companies developing airborne equipment and by the certification authorities.

DO-178B/ED-12 was published in 1992 by RTCA and EUROCAE (a non-profit organization addressing aeronautic technical problems). Written by a group of experts from aircraft and aircraft equipment manufacturing companies and from certification authorities, it provides guidelines for the production of software for airborne systems and equipment. The objective of the guidelines is to ensure that software performs its intended function with a level of confidence in safety that complies with airworthiness requirements.

DO-178B guidelines specify [23]:

- Objectives for software life-cycle processes.
- Description of activities and design considerations for achieving those objectives.
- Description of the evidence indicating that the objectives have been satisfied.

### 2.6.1 DO-178B development assurance levels

DO-178B defines five Development Assurance Levels [24]:

- Level A. Software that could cause or contribute to the failure of the system resulting in a catastrophic failure condition.
- Level B. Software that could cause or contribute to the system resulting in a hazardous or severe failure condition.
- Level C. Software that could cause or contribute to the system resulting in a major failure condition.
- Level D. Software that could cause or contribute to the system resulting in a minor failure condition.
- Level E. Software that could cause or contribute to the system resulting in no effect on the system.

### 2.6.2 DO-178B documentation requirements

DO-178B requires a thorough definition and documentation of the software development process. The base set of required documentation and life cycle artifacts include [25]:

1. Plan for Software Aspects of Certification (PSAC)
2. Software Quality Assurance Plan
3. Software Configuration Management Plan
4. Configuration Control Procedures
5. Software Code Standard For Ada
6. Software Design Standard
7. Software Requirements Standard
8. Software Development Plan
9. Software Verification Plan
10. Source, Executable Object Code, SCI and SECI
11. Software Design Document
12. Software Requirements Document
13. Traceability

14. Test Cases and Procedures
15. Verification Results
16. Quality Assurance Records
17. Configuration Management Records
18. Problem Reports
19. Software Accomplishments Summary

The section below mentions commonly occurring software defects and issues relating to software reuse.

## 2.7 Common software defects

Defects are inevitable in software. The Table 2.1 lists some common defects that are found in software. The list is not comprehensive but gives an idea of the types of errors [26]. Apart from these errors, there are possibilities of other errors which could be introduced in the system because of reusing of software.

| | |
|---|---|
| Crash-Causing Defects | • Null pointer dereference<br>• Use after free<br>• Double free |
| Incorrect Program Behavior | • Dead code caused by logical errors<br>• Uninitialized variables<br>• Erroneous switch cases |
| Performance Degradation | • Memory leaks<br>• File handle leaks<br>• Custom memory and network resource leaks<br>• Database connection leaks<br>• Mismatched array new/delete<br>• Missing destructor |
| Improper Use of APIs | • STL usage errors<br>• API error handling<br>• API ordering checks |
| Security Vulnerabilities | • Array and buffer overrun<br>• Unsafe uses of tainted data |
| Defects in Multithreaded systems | • Missing Synchronization points<br>• Starvation of a task<br>• Concurrency bugs<br>• Deadlocks<br>• Priority inversion |

**Table 2.1 Types of defects**

The effects of software reuse are discussed below.

### 2.7.1 Software reuse issues

Software reuse is very common across industries. This could be reusing a part of software that was initially developed for another application or it could be software developed for use by concurrent projects. Although software reusability is an advantage in many places there could be

places where software reuse could cause issues. This section describes a few issues concerned with software reuse.

In the traditional case, a single manufacturer typically develops and integrates the software. Objectives about requirements were developed with a particular system in mind. When multiple stakeholders become involved in the software assurance process, determining the types and levels of requirements may become more difficult. Objectives related to traceability and to compliance and consistency with system requirements is most likely not satisfied. This might cause trouble for getting the software certified by some standard.

Software reuse raises the question of re-verification. Re-verification activities depend on the situation (such as same or different processor, same or different compiler, same or different compiler options, and so on).

Reusable software often has functions in the initial application that will not be used in the subsequent applications. Hence deactivated code is a common reuse issue. Since it is developed for various applications, out-of-range data or unexpected input must be anticipated. Robustness of the software becomes questionable.

Initial efforts for the certification of reusable software are higher than the common certification of an application. Hence certification for a reusable software component is done only if the software does not change and is very often used like control algorithm.

Commercial-Off-The-Shelf (COTS) software is a special case of previously developed software, as it may get developed independently from the corresponding application. Operating systems, for example, is considered COTS. A separate software life cycle data is needed for COTS components too [27].

However, to detect errors which could be caused due to human negligence, verification could be effectively employed at various stages of the Software Development life cycle.

The section below explains the parameters for software certification.

## 2.8 Parameters for software certification

Software certifications are granted by Standards and Organization that considers the impact of the software based on its impact on the end user. Certified software is more trusted than the non-certified since they have more stringent requirements for the specification, development, testing, verification and maintenance. Based on the objectives in the standard, a process has to be established to ensure that the software is reliable. The sufficiency is defined by the circumstances and failure conditions of the software [28].

There are several parameters based on which the software is certified.

The following are some of the stages of the project that have to be considered in advance:

- Planning Process
- Development Process
    - Requirements Process
    - Design Process
    - Implementation Process
- Verification Process
- Configuration Management Process
- Quality Assurance Process
- Customer and Certification Liaison

It is generally required that most of the effort is spent on the requirements and verification phase of the software development lifecycle. This would require that requirement and design phase should try to address and capture the likely problems that would be faced during the course of the project. In the verification phase the use of automated test suite have to be considered to save a lot of effort and to eliminate human errors and false reports. Efforts should be put to arrive at a test suite that would be suitable for the current and future projects.

The design phase should list all the requirements, which have to be traceable to the high level requirements and the support the understanding of complex algorithms and data structures. The traceability should be able to trace all the requirements from design down to the test cases, including how they have been implemented in the source code. Each requirement has to be self-contained because this supports the verification of each requirement.

The requirement has to be checked for testability. The easiest way to handle this is to write functional test cases corresponding to the requirements, in parallel. This would identify testability problems much early.

Most effort is needed at the verification phase of the software life cycle. According to the paper on Parameters for Efficient Software Certification, the effort spent on verification is 35% whereas the effort for design is 25%, implementation is 20%, 10% for management and another 10% for miscellaneous activities [1]. Based on estimations, the effort for this part of the software life cycle is up to 50% of the overall project effort. Every function may be tested and additional high-level tests, which verify the required functionality, complete the test of the whole system. Additionally, boundary values are used as input to check for robustness. Validation phase ensures the quality of every generated artifact and provides the transitions to the next phases. A tool that would automatically verify test cases and structural coverage would aid in the verification process. A well setup process that mandates proper review criteria and checklists and use of informal reviews at various stages and continuous improvement of the process would help in reducing errors in the system [1].

The section below explains about System Software Safety and its development process.

## 2.9 System software safety

The information in this section has been derived from the FAA System Safety Handbook, Chapter 10, 2002 [29]. Before addressing the safety requirements for software, it is important to understand the significance and necessity of Software Safety and the ways in which software can "fail".

The following list provides the most common software failure mechanisms that should be evaluated during the safety analysis process. One might face such situations when the software:

- Fails to perform the required function or performs a function that is not required.
- Possesses timing and/or sequencing problems.
- Fails to recognize a self-destructive scenario that requires a corrective action or a safety-critical function that requires initiating the appropriate fault tolerant response.
- Produces the intended but inappropriate response to a hazardous condition.

The causes for such failure mechanisms could be attributed to the below mentioned reasons:

- Specification Errors - Omitted or improperly stated or incorrect specifications.
- Design and Coding Errors - Errors introduced by the programmer like incomplete/incorrect interfaces, timing errors, incorrect algorithms, logic errors, lack of self-test etc.
- Hardware/Computer Induced Errors - Errors caused due to power supply transients, bit errors due to interference.
- Documentation Errors - Errors caused inaccurate documentation of specifications, design/test requirements etc.
- Debugging/Software Change Induced Hazards

### 2.9.1 Software safety development process

There are five main steps to be followed as a part of process to ensure development of software that guarantees safety. The Figure 2.1lists the steps. Each of the steps is briefly explained in this section.

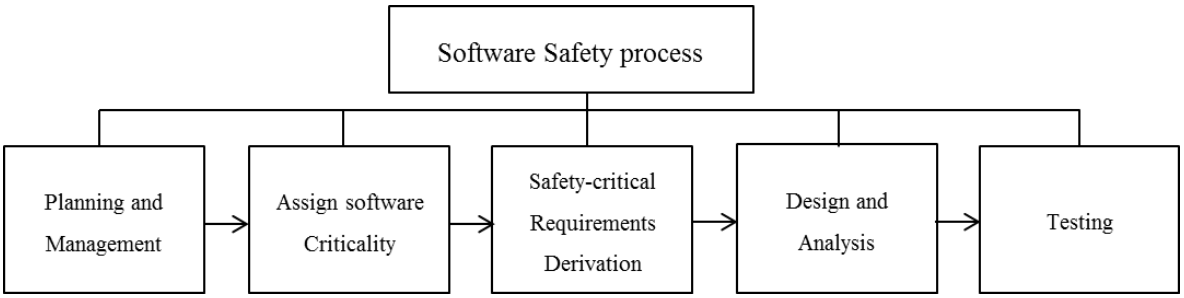


**Figure 2.1 Software safety process steps**

**Software safety planning and management**

Detailed planning and management helps in identifying critical software components and ensuring that all significant divisions related to the software are well connected. The list of checkpoints to be conformed to includes:

- Commissioning a safety team at the beginning of the software development life cycle.
- Defining acceptable levels of software risks consequently finding out necessary safety requirements.
- Understanding the scope of software along with its limitations and visualizing its interactions with the rest of the system's functions.
- Examining requirements, specifications and software application concepts to identify hazards/risks within safety critical software functions, hazardous commands, failure tolerance levels, sequence of events etc.
- Adhering to software safety standards throughout design and implementation by establishing appropriate verification and validation systems.
- Setting up testing plans and procedures to support software safety verification systems and thereof scrutinizing the results of such validation procedures until satisfactory.

As mentioned earlier, the software safety team has to be created first and they are entitled to handle the rest of the points stated. The Software Safety Working Group (SwSWG) is then established which has overall responsibility for the following:

- Monitoring and controlling the software safety program
- Identifying and resolving risks with software contributory factors
- Interfacing with the other Integrated Product Teams (IPTs), and
- Performing final safety assessment of the system (software) design.

**Assign software criticality**

Software failures are most commonly attributed to requirement or implementation phase errors wherein the bigger picture remains obscure. As a result, it is difficult to identify risks associated with the software and yet it is essential to categorize hazards and allocate sufficient resources to areas possessing highest risk potential. Hazard Risk Index (HRI) has been widely used to categorize hazards and the Software Hazard Criticality Matrix (SHCM) significantly helps in allocating safety requirements for various software modules between available resources and across architectures.

**Derivation of system safety-critical software requirements**

To derive system specific requirements, the overall list of requirements must be available. Further down, to identify system specific software safety requirements, there has to be a line of effect between the requirement and its associated risk that might pass down into requirements for subsystems.

**Design and analysis**

To reduce the safety risk of software responsible for safety-significant functions, it is primarily required to identify the hazards and failure modes of the system that are caused by the software or lack of software.

**Testing**

Two sets of analysis should be performed during the testing phase:

- Analysis before the fact to ensure validity of tests
- Analysis of the test results

The analysis of the validity of tests should ensure that the tests to be performed cover the intended safety critical functions and the analysis of the test results must guarantee that all safety requirements are satisfied. The satisfaction of safety requirements means that all the identified risks are either eliminated or at least controlled to an acceptable level of risk. The results of this analysis are handed over to the ongoing system safety analysis process. In some cases of high-risk software, an Independent Verification and Validation (IV&V) organization is also involved to oversee the software development including participation in validation of test analysis.

The System Safety Assessment Report (SSAR) is generally a Contract Data Requirements List (CDRL) item for the safety analysis performed on a given system. The SSAR shall contain a summary of the analyses performed and their results, the tests conducted and their results, and the compliance assessment.

The section below lists some software metrics used to measure size, complexity and the product quality.

## 2.10 Software metrics

The definition of software metrics given by Goodman is "The continuous application of measurement-based techniques to the software development process and its products to supply meaningful and timely management information, together with the use of those techniques to improve that process and its products" [30].

A significant number of software metrics have been developed in literature like Cyclomatic Complexity (CC) and Line of Code (LOC), and Object-Oriented metrics such as C.K metrics [31] and the metrics of Bansiya et al [32]. The metrics are broadly classified into two types:

- Design-based metrics
- Implementation-based metrics

**Design-based metrics**

These are metrics that can measure the software quality at the design phase. For example, Depth of Inheritance Tree (DIT) and Number of Children (NOC) metrics.

**Implementation-based metrics**

These metrics can be used for measure of the software quality only after the code is ready. For example, Cyclomatic complexity (CC) and Lines of Code (LOC).

**Correlation coefficient**

Correlation is also defined as a measurement of the strength of relationship between two random variables, and it predicts one variable of them from another [33].

Correlation between two variables X and Y is defined as:

$$\rho(X,Y) = \frac{COV(X,Y)}{\sqrt{Variance(X) * Variance(Y)}}$$

Where ρ (the Greek letter, rho) is called the Correlation Coefficient or the Product Moment Correlation Coefficient and Cov(X,Y) is called Covariance(X,Y) which is a measurement of linear dependence between two variables via a measurement of deviations from the means of both variables, and are in units of the x and y variables [14]. The Correlation Coefficient values range between -1 through 0 to +1.The relationship is stronger if the value lies closer to -1 or +1 [33] [34]. Correlation could mean either Positive, Negative or Zero correlation. Figure 2.2 shows an indication of the correlations [35].

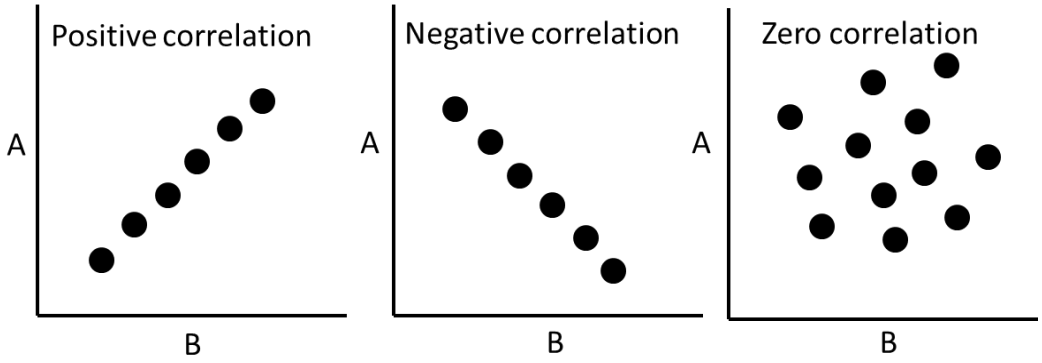


**Figure 2.2 Indication of the correlations**

The subsections below explain about the some commonly used software metrics for the code size, contents, software complexity and product quality.

## 2.10.1 Lines Of Code (LOC)

There are several ways to measure software size; a metric based on source lines of code (SLOC) is simply a count of the number of lines in a program's source code, typically adjusted for logical statements rather than physical lines. When further adjusted to remove comment lines, the metric is non-comment source lines (NCSL). NCSL is not a good measure of functionality or complexity; short software components can be very complex, and long software components can be very simple. However it can be used to identify trends and roughly quantify growth rate [36].

NASA's study on flight software complexity shows that there has been an increasing growth in the size of flight software for human missions and robotic missions from 1968 through 2005 and the trend has shown an exponential growth rate of a factor of 10 approximately every 10 years [36], which conforms to the Augustine's law that "software grows by an order of magnitude every 10 years" [37]. The study also shows that the flight software sizes for selected GSFC missions from 1997 through 2009, with NCSL ranges from 28,400 to 149,500 and for APL missions from 1995 through 2007, with the larger missions in the range of 100,000 lines of code [36].

With the increasing capability of microprocessors, adding more functionality has become very cost effective. As a result, the number of lines of code in the software is increasing. There has been a study that shows that the lines of code in a typical GM car was approximately 100000

KLOC in 2010 from 100 KLOC in 1970; while all of the codes are not critical to the car's operation, it is indicative of the continuing trend with embedded systems.

Although Line count is a good indicator of growth it does not explain about the difficulty level of the code or algorithmic complexity in the code.

### 2.10.2 Function points

A function is a collection of executable statements that performs a certain task, together with declarations of the formal parameters and local variables manipulated by those statements [38].

The essential concept behind the function point metric is to base the size of applications on external characteristics that do not change because of the programming language or languages used. In essence, a function point consists of the weighted totals of five external aspects of software applications:

- Number of external inputs × 4
- Number of external outputs × 5
- Number of logical internal files × 10
- Number of external interface files × 7
- Number of external inquiries × 4

There are also low and high weighting factors, depending on the complexity assessment of the application in terms of the five components [39] [40]:

- External input: low complexity, 3; high complexity, 6
- External output: low complexity, 4; high complexity, 7
- Logical internal file: low complexity, 7; high complexity, 15
- External interface file: low complexity, 5; high complexity, 10
- External inquiry: low complexity, 3; high complexity, 6

With the weighting factors, the first step is to calculate the function counts (FCs) based on the following formula:

$$FC = \sum_{i=1}^{5}\sum_{j=1}^{3} w_{ij} * x_{ij}$$

where $w_{ij}$ are the weighting factors of the five components by complexity level (low, average, high) and $x_{ij}$ are the numbers of each component in the application.

The second step involves a scale from 0 to 5 to assess the impact of 14 general system characteristics in terms of their likely effect on the application [41]. The 14 characteristics are:

1. Data communications
2. Distributed functions
3. Performance
4. Heavily used configuration
5. Transaction rate
6. Online data entry
7. End-user efficiency
8. Online update
9. Complex processing
10. Reusability
11. Installation ease
12. Operational ease
13. Multiple sites
14. Facilitation of change

The scores (ranging from 0 to 5) for these characteristics are then summed, based on the following formula, to arrive at the value adjustment factor (VAF)

$$VAF = 0.65 + 0.01\sum_{i=1}^{14} c_i$$

where $c_i$ is the score for general system characteristic i. Finally, the number of function points is obtained by multiplying function counts and the value adjustment factor:

$$FP = FC * VAF$$

This equation is a simplified description of the calculation of function points. The International Function Point User's Group Standard should be consulted for a complete treatment [42]. Function points provide better than the older lines-of-code-metrics. Table 2.2 below shows the approximate sizes of some software applications [43]. Sizes based on IFPUG version4 and SPR logical Statement Rules.

| Application | Type | Purpose | Primary Language | Size, KLOC | Size, FP | LOC per FP |
|---|---|---|---|---|---|---|
| Graphics Design | Commercial | CAD | Objective C | 54 | 2700 | 20.00 |
| Visual Basic | Commercial | Compiler | C | 375 | 3000 | 125.00 |
| IMS | Commercial | Database | Assembly | 750 | 3500 | 214.29 |
| Knowledge Plan | Commercial | Project Management | C++ | 134 | 2500 | 56.67 |
| WMCCS | Military | Defense | Jovial | 18000 | 175000 | 102.86 |
| Aircraft Radar | Military | Defense | Ada 83 | 213 | 3000 | 71.00 |
| Gun Control | Military | Defense | CMS2 | 250 | 2336 | 107.00 |
| Airline | MIS | Business | Mixed | 2750 | 25000 | 110.00 |
| Windows XP | Systems | Operating System | C | 25000 | 85000 | 129.41 |

**Table 2.2 Approximate sizes of selected software applications**

### 2.10.3 Complexity

The IEEE Standard Computer Dictionary defines 'complexity' as "the degree to which a system or component has a design or implementation that is difficult to understand and verify" [44].

In a study of failures in complex systems, Dietrich Dörner, a cognitive psychologist, provides perhaps the best definition of complexity [45].

"Complexity is the label we give to the existence of many interdependent variables in a given system. The more variables and the greater their interdependence, the greater that system's

complexity. Great complexity places high demands on a planner's capacities to gather information, integrate findings, and design effective actions. The links between the variables oblige us to attend to a great many features simultaneously, and that, concomitantly, makes it impossible for us to undertake only one action in a complex system. … A system of variables is 'interrelated' if an action that affects or is meant to affect one part of the system will also affect other parts of it. Interrelatedness guarantees that an action aimed at one variable will have side effects and long-term repercussions."

There are some metrics that are used to measure complexity of software. Table 2.3 mentions the metrics and their purpose.

| Complexity Metrics | Primary measure of … |
|---|---|
| Cyclometric complexity (McCabe) | Soundness and confidence; number of linearly- independent paths to a program module; strong indicator of testing effort |
| Halstead complexity Measures | Algorithmic complexity, measured by counting operators and operands; a measure of maintainability |
| Henry and Kafura metrics | Coupling between modules (parameters, global variables, calls) |
| Bowles metrics | Module and system complexity; coupling via parameters and global variables |
| Troy and Zweben metrics | Modularity or coupling; Complexity of structure (maximum depth of structure chart); calls-to and called-by |
| Ligier metrics | Modularity of structure chart |

**Table 2.3 Software complexity metrics**

### Cyclometric complexity

Cyclomatic complexity is often referred to as the McCabe number. The complexity of a program unit or a module by measuring the amount of decision logic is determined by McCabe's model [46]. It is a measure of number of linearly- independent paths to a program module. It predicts reliability of the software.

The cyclomatic complexity metrics are described below. For any given computer program, its control flow graph, G, can be drawn. Each node of G corresponds to a block of sequential code and each arc corresponds to a branch of decision in the program. The cyclomatic complexity [9] of such a graph can be computed by a simple formula from graph theory, as:

$$V(G) = E - N + 2P$$

where v(G) = cyclomatic complexity
E = the number of edges of the graph
N = the number of nodes of the graph
P = the number of connected components.

Cyclomatic complexity v(G), that measures the number of logical paths in a module, is a size indicator. It is the minimum number of tests needed to forecast high reliability. Modules with numbers below a threshold of 10 are considered reliable. As the number of branching into and branching out structures increases, maintaining code becomes difficult [47].

### 2.10.4 Product quality metrics

The most commonly used product quality metrics are Defect Density and Mean Time To Failure (MTTF).

**Defect density**

Defect density is defined as:

$$\text{Defect Density} = \frac{\text{Number of Defects Found}}{\text{Size}}$$

The more common way of defining defect density is in terms of delivered defects to the customer (i.e., post-release defects). The size of the system can be measured in many different ways. The most common are thousands of lines of code (KLOC) or Function Points (FP) [48].

Defect density can be localized to a specific phase. For example, defect density for the testing phase. The Table 2.4 below shows the number of post-release defects per thousand lines of code

(KLOC) in various industries in the U.S and Canada from Rubin, H. and E. Yourdon (1995). Industry Canada Worldwide Benchmark Project, Industry Canada.

| Business Domain | U.S | Canada |
|---|---|---|
| Aerospace | 2.49 | 4.8 |
| Financial | 3.1 | 4.2 |
| Software | 2.8 | 2.02 |
| Distribution | 2.1 | N/A |
| Telecommunication Equip. | 1.8 | 15 |
| Telecomm. Services | 1.9 | 8.5 |

**Table 2.4 Number of post-release defects per KLOC**

Table 2.5 shows delivered defects per Function point for various business domains. For MIS and Commercial projects, a small project is 100 FP, a medium project is 1000 FP and a large project is 10000 FP. For systems software and military projects, a small project is 1000 FP, a medium project is 10000 FP and a large project is 100000 FP. The data is derived from Jones, C. (2000). Software Assessments, Benchmarks, and Best Practices, Addison-Wesley [49].

| Business Domain | Small Projects | | Medium Projects | | Large Projects | |
|---|---|---|---|---|---|---|
| | Average | Best | Average | Best | Average | Best |
| MIS | 0.15 | 0.025 | 0.588 | 0.066 | 1.062 | 0.27 |
| Systems Software | 0.25 | 0.013 | 0.44 | 0.08 | 0.726 | 0.15 |
| Commercial | 0.25 | 0.013 | 0.495 | 0.08 | 0.792 | 0.208 |
| Military | 0.263 | 0.013 | 0.518 | 0.04 | 0.816 | 0.175 |

**Table 2.5 Delivered defects per function point**

### Mean Time To Failure (MTTF)

Mean Time To Failure (MTTF) measures the time between failures. MTTF is most often used with safety critical systems such as the airline traffic control systems, avionics, and weapons. For example, the U.S. government mandates that its air traffic control system cannot be unavailable for more than three seconds per year. In civilian airliners, the probability of certain catastrophic failures must not be worse than $10^{-9}$ per hour [41] [50].

The Table 2.6 below illustrates the minimum, average and maximum Mean Time To Failure (MTTF) for commercial, defense/aerospace and semiconductor equipment industries. The "initial" milestone is the first month of delivery. The "final" milestone is the last month before the version is updated with a major release. Between the initial and final releases there are one or more minor maintenance releases [51].

| Milestone | Defense | Semiconductor | Regulated Commercial | Commercial/ not regulated |
|---|---|---|---|---|
| Actual MTTF initial (hours) | 102 | 148 | 423 | 24 |
| Actual MTTF final (hours) | 853 | 360 | 3285 | 243 |

**Table 2.6 Average MTTF values by industry**

The sections below explain about Concolic Execution and the working of some tools which use this technique.

# 2.11 Concolic execution

There are always several branches in the code which together form different execution paths. By using manually generated test cases, it is highly possible that there are some parts of the code that have not been exercised or there are some paths that are difficult to reach. Such unexercised code can cause failure of the software. So it is important that all possible paths in the code have been exercised to test all possible behaviors of the software program.

Code coverage is a very important parameter of software test cases. There exist a number of search strategies that can be used for the exploration. Symbolic execution based methods are

gaining popularity for automated test generation of software. Concolic stands for the simultaneous concrete and symbolic execution of a program.

The paper “Heuristics for Scalable Dynamic Test Generation” by Jacob Burnim and Koushik Sen from EECS, UC Berkeley, deals with few concolic search strategies, namely Bounded Depth-First Search, Control-Flow Graph Directed Search, and Uniform Random Search. The strategies have been implemented using CREST, an open source automatic test generation tool for C [52].

The program under test is first instrumented to include function calls which are required to be executed symbolically. The original code of the program performs the concrete execution and the inserted function calls perform the symbolic execution. Concrete values can be used to simplify complex constraints. The conditions in the code are extracted and converted into a formula which is basically constraints for the particular path that was executed. According to the strategy, some condition in the previous path constraint is reversed. This new path constraint is then given to a Constraint solver. A popular choice is the Satisfiability Modulo Theory (SMT) solver. The SMT solver generates inputs that exercise this new path or it declares the path as infeasible.

In the formula, the branch predicates are represented in single static assignment (SSA) form where each variable would be assigned exactly once. If there is more than one assignment on a variable, new version of the variable would be created.

Consider the code snippet below. The numbers at the beginning of each line is the line number.

```
0 void function1(int a, int b){
1       b = b + 1;
2       c = 5 * a;
3       if(a != b){
4               if(c == a + 20){
5                       abort();
6               }
7       }
8       return;
9  }
```

The branch predicates are computed in terms of the symbolic variables defined in place of the actual program variables [53]. For executing the path consisting of lines 0,1,2,3,8, the branch constraints would be:

***b1 = b0 + 1***

***c0 = 5.a0***

***a0 != b1***

***c0 != a0 + 20***

The path predicate for the path 0 ->1 -> 2 -> (3; true) -> (4; false) is computed with the help of the branch constraints of the path as: (b1 = b0 + 1) ^ (c0 = 5.a0) ^ (a0 != b1) ^ (c0 != a0 + 20). This, in turn, can be represented in terms of the input variables via symbolic simulation as: (a0 != b0 + 1) ^ (5a0 != a0 + 20).

Using the basic depth-first search strategy in the context of the C program above, let's assume that a symbolic test generation engine initially generates the value 1 for a0 and 0 for b0. As a result, 'function1' executes the false branch of the first if-statement and hence has the path constraint (a0 = b0+1). To force the execution to the true path, it tries to negate the constraint of the last execution. The generated formula would be fed as input to a SMT solver, which might give an output a0 = 0 and b0 = 0.

Executing again with these inputs we execute the path 0 ->1 -> 2 -> (3; true) -> (4; false) forming path constraints (a0 != b0 + 1)^(5a0 != a0 + 20). To force the execution to the true path, it tries to negate the constraint of the last execution. The formula now generated would be (a0 != b0 + 1)^(5a0 = a0 + 20). On solving this new formula, we might get the output a0 = 5, b0 = 0. The execution now reaches the abort() statement. This procedure iterates until all possible branches in the function have been covered. This procedure uncovers the bugs as the paths of the program are explored in a depth-first manner. Since all possible paths are explored automatically, this verification technique would guarantee effectiveness irrespective of the Software Developer's skills.

## 2.12 KLEE

KLEE [54] is an automatic test generation program for C which is based on Symbolic and Concrete execution. KLEE is a base for several tools such as KLOVER and Cloud9. It employs some constraint solving optimizations and search heuristics to obtain high test coverage. It works by substituting program inputs with symbolic values. When the program branches based on a symbolic value, the systems traces both the branches maintaining a separate path condition for every path traces. When a path terminates or a bug is found, it generates a test case by solving the path constraints to obtain concrete values. The path followed by KLEE is always the same as the path followed by the original path. This makes sure that there are no false positives. The code is compiled into byte codes using LLVM [55] compiler for GNC C. KLEE uses STP as its constraint solver.

### 2.12.1 KLEE architecture

KLEE, at a high level, works like a hybrid between an operating system for symbolic processes and an interpreter with every symbolic process having a separate register file, stack, heap, program counter and path condition. In KLEE symbolic process is referred to as a state. LLVM assembly language has a RISC-like virtual instruction set. KLEE interprets this instruction set and maps the instructions to constraints without approximation (i.e. bit-level accuracy).

KLEE has an interpreter loop as its core which selects a state (from a large number of states) to run and then symbolically execute a single instruction in the context of that state. This continues until all the states have been traversed or the user time out is reached.

Conditional branches take a branch condition which is a Boolean expression and change the instruction pointer of the state based on the condition. The query is solved by the constraint solver to determine if the branch condition is provably true or false and the instruction pointer is changed to take the appropriate location. If both branches are possible, then the state is cloned to explore both the paths.

At every operation that could lead to an exception, branches are generated to check if there exists an input value to cause an error. For example, division instruction generates a branch to check for divide by zero error and load or store instructions are checked for out of bound memory errors. If a pointer can refer to many objects, KLEE clones the current state to match the number of objects and then performs the appropriate operation.

### 2.12.2 Search strategies

When there is more than one active path, KLEE uses two search heuristics to pick the first one to execute:

- Random path selection
- Coverage-optimized search

*Random path selection* records the path followed for all active states using a binary tree. The current states are leaves of the tree and the internal nodes represent forked executions. States are selected randomly by traversing this tree from the root and selecting the path to follow at branch points.

*Coverage-optimized search* tries to select states likely to cover new code in the immediate future. It uses heuristics to compute a weight for each state and then randomly selects a state according to these weights.

These strategies are used in a round robin fashion. Although this might be time consuming in few cases, it can be used to prevent cases where a single strategy gets stuck. This interleaving technique can improve the overall effectiveness.

### 2.12.3 Query optimizations

The following are the main query optimizations done to reduce the number of states and to reduce the time taken to arrive at a test case:

- **Expression rewriting**

  For example, simple arithmetic simplifications ($x + 0 = x$)

  Strength reduction ($x * 2^n = x << n$)

  Linear simplification ($2*x - x = x$)

- **Constraint set simplification**

  KLEE actively simplifies the constraint set by rewriting previous constraints when new equality constraints are added to the constraint set. For example, an inexact constraint such as $x < 5$ gets added, followed some time later by the constraint $x = 2$. By substituting the x in the constraint it gets simplified to true and is eliminated from the constraint set.

- **Implied value concretization**

  For example, when a constraint ($x + 1 = 5$) is added to the path constraint, it is implicitly concretized by making x as 4.

- **Constraint independence**

  KLEE eliminates irrelevant constraints before sending it the solver by dividing the constraint sets into disjoint independent subsets based on symbolic variables they reference. For example, given the constraint set $\{i < j, j < 15, k > 0\}$, a query of whether $i = 15$ just requires the first two constraints.

- **Counter-example cache**

  The counter-example cache maps sets of constraints to counter-examples along with a special sentinel used when a set of constraints has no solution. This mapping is stored in a custom data structure for efficient searching for cache entries for both subsets and supersets of a constraint set.

# 2.13 KLOVER

KLOVER [56] is the first symbolic execution and automatic test generation tool for C++ programs. It is built on top of KLEE, a symbolic execution engine for handling C programs. The tool has provisions for handling C++ features, and optimizations necessary for more efficiency and scalability. A C++ program is compiled into LLVM [55] byte code, which is interpreted by KLOVER. After symbolic execution, it provides a set of outputs, which are concrete test cases. This output can be replayed in the real setting using a compiler like GCC and run on a machine. gcov [57] is used to produce the source program coverage. Figure 2.3 shows the architecture of KLOVER [56].

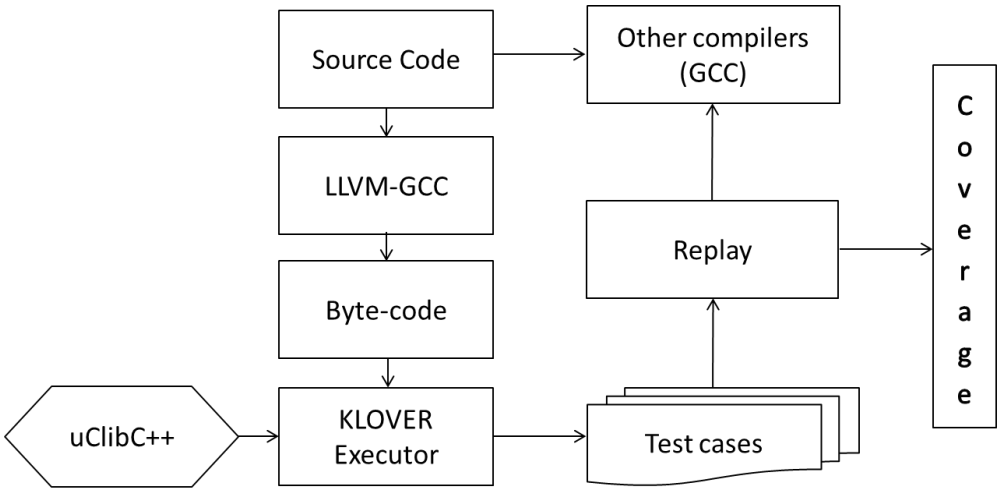


**Figure 2.3 Architecture of KLOVER**

A symbolic state in KLOVER models a machine execution state. A register stores either a concrete value or a symbolic expression. A memory, which is organized as components, has a concrete address and an array of bytes recording the value. KLOVER supports automatic dynamic resizing since the memory blocks of different objects do not have to be consecutive. A new state is generated for each reference of a pointer referring to multiple components, which is determined by the SMT solver.

KLOVER extends the LLVM instructions and external functions to support Advanced Instructions, Exceptions, C++'s Run-time Type Information (RTTI) and changes to Memory model. To improve the performance of symbolic execution, thee uClibc++ library is optimized [58] which is compiled into the LLVM byte code and dynamically loaded into the engine. Two versions of C++ libraries are maintained: one for symbolic execution and one for concrete values and JIT compilation of external functions.

Several other optimizations have been written to avoid unnecessary conditional statements to reduce the number of generated paths, to convert expensive expressions into cheaper ones, and to build fast decision procedures into the library implementation. KLOVER has support for declaring arrays with symbolic length and declaration of a possibly null pointer to reduce the burden of manual testing.

## 2.14 Cloud9

Cloud9 [59] is a testing platform that is built on KLEE. It bridges the gap between symbolic execution and the requirements of the automated testing in the real world. By including a new symbolic environment model that supports all major aspects of the POSIX interface such as processes, threads, synchronization, networking, IPC and file I/O it has an ability to handle multi-threaded and distributed systems. It parallelizes the symbolic execution in a way that scales well on large clusters. It provides a solution to handle the challenges of path explosion and ways to mediate between program and its environment.

### 2.14.1 Design of Cloud9

In a classical symbolic execution model, a program has unconstrained symbolic input, which is executed by a symbolic execution engine (SEE). When it encounters a branch that involves

symbolic values, execution is forked into parallel executions with each branch maintaining its own clone of the current state. Symbolic values in the clones are constrained to make the branch condition evaluate to false (e.g., p >= MAX) respectively true (e.g., p <MAX). This repeats recursively at every subsequent branch forming an execution tree of the otherwise linear execution program. This way all the possible execution paths are explored.

Consider the small example code snippet below. Figure 2.4 shows how an execution tree is formed for the code.

```
void function1 ( int p )
{
        if (p < MAX)
        {
                if (p > 0)
                ...
                else
                ...
        }
         else
        {
                if (p > 7)
                ...
                else
                ...
        }
}
```

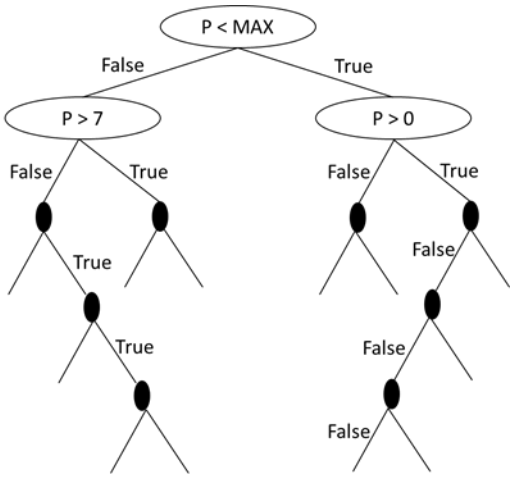


**Figure 2.4 Execution tree for symbolic execution**

As the number of branches increase, the size of the execution tree increases with the constraints becoming more complex. With parallel execution cloud9 can enable individual cluster nodes that can explore the execution tree independent of each other. Cloud9 partitions the tree dynamically as the tree is explored. For this cloud9 consists of worker nodes and a load balancer (LB).

### Dynamic distributed exploration

Each worker node can run independent symbolic execution engine based on KLEE consisting of a runtime and a searcher and a constraint solver. They explore parts of the execution tree and send their statistics to update the LB, which instructs pairs of workers to balance each other's on a needed basis. The LB is not involved in the critical path since the encoding and transfer of work is handled by the workers directly. Figure 2.5 explains the architecture of cloud9 [59].

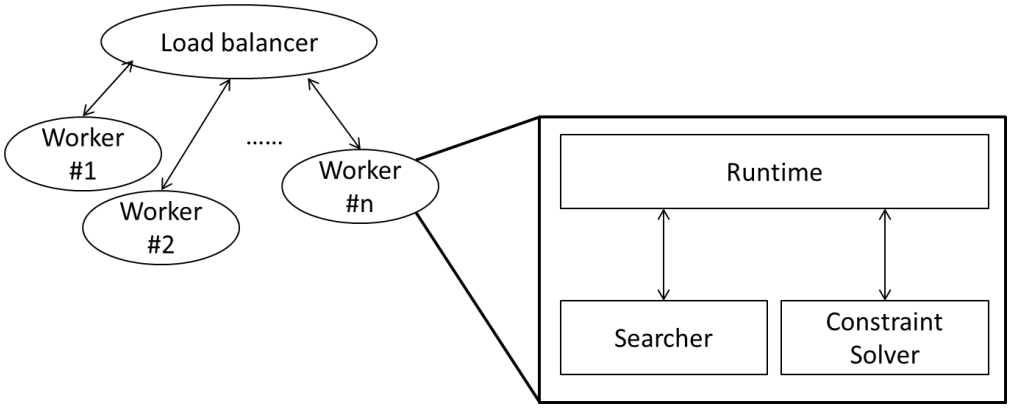


**Figure 2.5 Architecture of Cloud9**

The Load balancer is the first component to come up. As and when there are new workers available, the LB balances the load with the new worker based on the loads of other workers. The workers send updates to the LB regarding their progress in terms of code coverage encoded as bit vectors. The locally updated coverage is then updated by the LB by ORing it with the global coverage that it maintains and the result is then sent back to the workers to update its local copy of the coverage.

The balancing algorithm ensures that the worker queue lengths stay within the same order of magnitude. It takes as input the lengths $l_i$ of each worker $W_i$'s queue $Q_i$. It computes the average $\bar{l}$ and standard deviation $\sigma$ of the $l_i$ values and then classifies each $W_i$ as under loaded ($l_i < \max\{\bar{l} - \delta \cdot \sigma, 0\}$), overloaded ($l_i > \bar{l} + \delta \cdot \sigma$) , or OK otherwise; $\delta$ is a constant factor. The $W_i$ are then sorted according to their queue length $l_i$ and placed in a list. LB then matches under loaded workers from the beginning of the list with overloaded workers from the end of the list. For each pair $\langle W_i, W_j \rangle$ with $l_i < l_j$, the load balancer sends a job transfer request to the workers to move $(l_j - l_i)/2$ candidate nodes from $W_j$ to $W_i$. Although LB issues the transfer request to pairs of workers, it is the source node that decides the job to be transferred.

No element in the system maintains the global execution tree. The parts of the sub trees are disjoint to ensure that there is no redundancy of work and together they cover the entire global execution tree. The worker consists of three types of node:

- Dead nodes

  Nodes that have already been explored and are thus not of much interest.
- Fence nodes

  Nodes that delineate the portion being explored and separates the domain of other workers.
- Candidate nodes

  Nodes those are ready to be explored. They are the leaves of the local tree and form the exploration frontier

The union of all the local frontiers forms the frontier of the global execution tree.

**Lifecycle of a node**

A node N in $W_i$'s subtree has two attributes $N_{status}$ ε {materialized, virtual} and $N_{life}$ ε {candidate, fence, dead}. A worker's frontier $F_i$ is the set of all candidate nodes on worker $W_i$. The worker can only explore nodes in $F_i$, i.e., dead nodes are off-limits and so are fence nodes, except if a fence node needs to be explored during the replay of a job path. The union ⋃ $F_i$ equals the frontier of the global execution tree, ensuring that the aggregation of worker-level explorations is complete. The intersection ∩ $F_i$ = Ø, thus avoiding redundancy by ensuring that workers explore disjoint sub trees. Figure 2.6 summarizes the lifecycle [59].

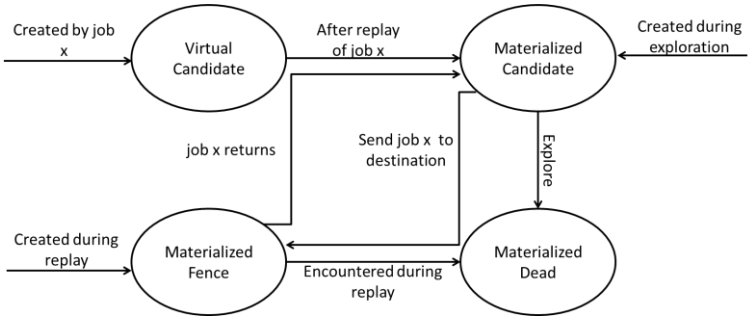


**Figure 2.6 Lifecycle of a node**

**POSIX model**

Cloud9 links the program with uClibc library, parts of which was replaced with POSIX model code. This section briefly explains the architecture of cloud9 POSIX model. The operations related to threads, processes, file descriptors, and network were replaced with the models and the APIs of which were augmented with Cloud9-specific extensions. Symbolic System calls interface the POSIX components with Symbolic execution Engine. To support POSIX model, feature for multiple address spaces per state and support for scheduling threads and processes were added to KLEE. The POSIX model uses shared memory to track all system objects. System buffers model half-duplex communication channels supporting event notification to multiple listeners. Block buffers are random-access, fixed-size buffers. Their operations are not blocked. They are used to implement symbolic files. Cloud9 implements a cooperative scheduler.

The next section explains about the tool VCC.

## 2.15 Verifier for Concurrent C (VCC)

VCC [60] [61] is a mechanical verifier for concurrent C programs. VCC takes a C program, annotated with function specifications, data invariants, loop invariants, and ghost code, and tries to prove these annotations correct. Using these specifications it verifies one function/type definition at a time that makes it modular and thus helps in verifying functions independently. This makes the tool highly scalable. It can be used for programs that are concurrent. VCC has support for low-level features like bit fields, unions, and wrap-around arithmetic.

VCC is integrated in Microsoft Visual Studio, allowing developers to verify code from within their normal work environment while working on that code. It comes with support for verification and debugging. It has an explorer for counterexamples of failed proofs that can be used to find errors in code or specifications. It provides a prover log analyzer that aids in debugging. VCC is currently used to verify the core of Microsoft Hyper-V, consisting of 50,000 lines of system-level C code.

When successful, VCC promises that your program actually meets its specifications. VCC extends C with design by contract features, like pre and post condition as well as type invariants. Annotated programs are translated to logical formulas using the Boogie tool, which converts them into BoogiePL [62], which is an intermediate language for verification purposes, and passes them to an automated SMT solver Z3 [63] to check their validity. To disprove a condition, it generates a counter-example, which consists of a sequence of program states with variable assignments. This can be easily viewed using VCC's Model Viewer. It shows the trace through the method that leads to the violation of the verification condition; in addition it shows the memory and additional attributes on the memory for each different state of this trace.

Figure 2.7 explains the workflow of VCC with a flow diagram [61].

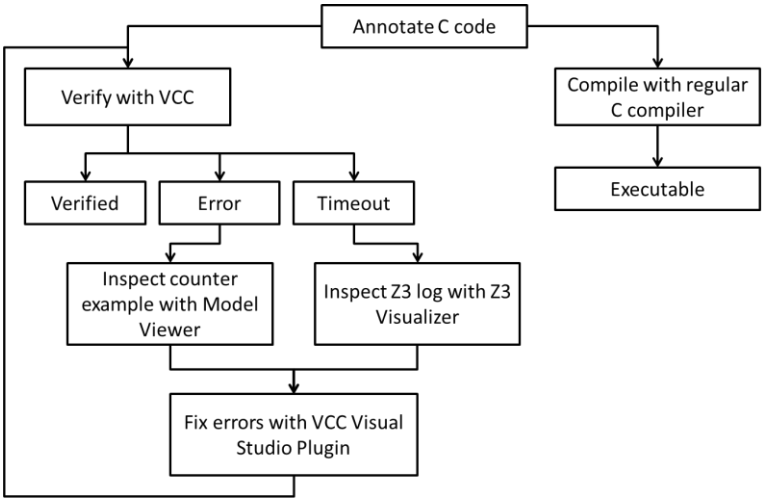


**Figure 2.7 VCC workflow diagram**

VCC flags errors like arithmetic overflows, and weakens intermediate assertions to compensate for interference from other threads apart from asserting type-safe memory access.

VCC uses a typed memory model [64]. It significantly reduces both the burden on the annotator and the theorem prover. It uses a ghost state to keep track of the invariants, which describe the program properties, and to maintain the type information.

**Two state invariant**

VCC can be used to verifying the implementation of lock-free data structures that are commonly used for thread synchronization, like spinlocks or reader/writer locks by using two-state invariants that allows atomic modification of shared objects. By using two-state invariant, pre-state can be referenced which can be used to constrain atomic changes.

**Ownership**

It uses a process called opening and closing of objects to temporarily allow multiple changes to the object that are exclusively owned by a thread, by disabling the object's invariant before the invariant can be reestablished. To modify a shared object the thread has to take the ownership of the object from another object, open it, modify and then close it before transferring the ownership to another object or thread.

**Claims**

Object invariants hold only when an object is closed. Hence, useful shared state information can be obtained only from objects that are known to be closed. To guarantee this VCC defines Claims. Claims are handles that prevent other objects from being opened, thus allowing the owner of the claim to rely on the claimed object's invariant. Claims can be created and destroyed dynamically.

The next chapter explains in detail on how to use Cloud9 to verify the code and extract invariants.

# 3. Verification using Cloud9

This chapter explains in detail about the program that was verified using Cloud9, the various steps involved in extracting invariants for the program and how they were analyzed and steps that were followed to verify the code using cloud9.

The only input from the AUV team was the pseudo-code that needed to be verified. The first step was to understand the pseudo-code and its functionality before actually proceeding to the implementation of the code. The section below explains the pseudo-code structure for the AUV depth controller.

## 3.1 Pseudo-code for an AUV depth controller

The pseudo-code for the AUV depth controller gives a picture of the code implementation and the structure. There are several processes that handle the device I/O, process the incoming data, and a process that monitors the parameters for any safety violation. All of these processes need to run in parallel. The following are the main processes that run in the AUV depth controller.

- Orientation sensor (AHRS)
- Depth sensor
- Fin actuator
- Depth controller (PID)
- Pitch controller (State space)
- Depth monitor

Each of these processes performs a task of either monitoring the data using sensors or processing the data that is coming in from one of the sensors. Essentially these processes behave like a network of nodes that send and receive data. The functionality of the various processes are explained in section 3.1.1 below.

### 3.1.1 Functionality of the parallel processes in pseudo-code

**Orientation sensor (AHRS)** reads rate *f*, serial port *SP*, baud rate *B* from configuration file, opens serial port *SP* at baud rate *B* and command AHRS to transmit continuously at rate *f*. In a loop, it keeps reading the serial port until response header byte *R* is found, reads *NR* bytes associated with response *R*, parses orientation *O* from response *R* (message is array of IEEE 32-bit floats) and publishes the orientation *O* to any subscribers.

**Depth sensor** reads rate *f*, serial port *SP*, baud rate *B* from configuration file, opens serial port *SP* at baud rate *B*. In a loop, it reads serial port until message delimiter is found (\n), extracts from buffer since last message until this message delimiter into message string *M*, parses pressure *P* from message *M* (message is ASCII float), converts pressure *P* to depth *D* and publishes depth *D* to any subscribers.

**Fin actuator** reads serial port *SP*, baud rate *B* and servo constant $K_{servo}$ from configuration file, opens serial port *SP* at baud rate *B*. In a loop calculates servo command $S = K_{servo} * \delta_{fin}$ and writes servo command *S* to serial port *SP*.

**Depth controller (PID)** reads rate *f*, controller gains *KP, KI, KD* from configuration file and a desired depth $D_{des}$ from user and subscribes to depth *D*. In a loop, it reads the most recent depth *D* and calculates depth error which using the formula

$$E = D_{des} - D$$

In the first iteration, it sets error integral *EI* and error derivative *ED* to zero. From the second iteration EI and ED are calculated as below:

$$EI = EI_{last} + E / f$$

$$ED = (E - E_{last}) * f$$

It calculates the desired pitch according to formula

$$P_{des} = KP*E + KI*EI + KD*ED$$

and publishes the desired pitch $P_{des}$.

**Pitch controller (State space)** reads rate *f*, controller matrices *A, B, C, D* from configuration file, computes period $T = 1/f$ , subscribes to orientation *O* and desired pitch $P_{des}$. In a loop, reads most recent desired pitch $\mathbf{P_{des}}$, Orientation values *O* and $O_{rate}$ and sets the pitch error *E* according to the formula

$$E = P_{des} - P$$

In the first iteration it sets the error Integral *EI* to zero and from the second iteration calculates the *EI* according to the formula

$$EI = EI_{last} + E / f$$

It forms the input vector $u = [O_{rate}\ E\ EI]^T$ and updates the controller states *x* according to the formula

$$x = A * x_{last} + B * u$$

and calculates fin angle as

$$\delta_{fin} = C * x + D * u$$

and publishes fin angle $\delta_{fin}$.

**Depth monitor** reads maximum depth $D_{max}$ from configuration file and subscribes to depth *D*. In a loop at a fixed rate f checks if *D* has been updated in the last period *1/f*. If not it exits. It also monitors to check if the depth has crossed the maximum depth $D_{max}$ and exits if it has crossed the maximum depth.

A careful observation of the pseudo-code reveals a trend where there are some processes which publish information that are obtained by other processes which subscribe to the specific information. It can also been seen that there is only a single specific location in the entire system where every parameter is updated. This can be correlated with the Single Static Assignments

(SSA) where every variable is updated only once in the code which makes it convenient to apply symbolic and concrete execution which uses SSA.

## 3.2 Simulating the program

After the pseudo-code was understood, the next step was to implement the program that simulates the pseudo-code. The code was written in C++ with the use of POSIX threads. The implemented code was about 700 lines.

Each of the parallel processes mentioned in the pseudo-code corresponds to a separate thread. Being a multi-threaded environment there are possibilities of several errors. Such errors may include

- Missing Synchronization points,
- Concurrency bugs (Example, Consumer trying to read value before it is published by producer), and
- Starvation of task (or missed deadline) due to high priority tasks or interrupts are some common bugs that can be found in multi-threaded programs.

The program ensures that there are no concurrency bugs by preventing read before write and there are no simultaneous read and write to the same variable by appropriately using mutex and spinlocks.

Since each of these threads continue performing a particular task that involves a loop, as mentioned in the pseudo-code above, exit condition of the threads becomes critical. For the purpose of AUV depth controller, it is assumed that there are no hardware failures and the data provided sensors are valid. All threads exit whenever there is any violation seen by the Depth

monitor thread that checks if the current depth is greater than the maximum depth and if the depth has not changed over a period of time.

## 3.3 Invariant extraction

Extracting invariants from the code is one of the preliminary measures with this method of verification. There are two principal challenges in extracting the invariants: choosing what invariants to infer and performing the inference. To tackle the first challenge, understanding the code and the functions become very critical. The code was analyzed to ascertain the parameters of interest. These parameters would be a part of the invariants. From the AUV depth controller code five parameters of interest were identified.

The following are the parameters identified:

- desired pitch
- fin angle
- orientation
- depth
- servo command

The values of these parameters need to be monitored for every possible value during the simulation of the code. As the initial step towards invariant extraction, the code was instrumented with print statements in all different thread functions that print the values of these variable (or parameters) for every execution of the thread to a file. These prints act as traces of the program execution. The technique is to execute a program on a collection of inputs and infer invariants from captured variable traces. We note that other techniques to capture the values of the variables without printing them are possible.

After the traces were captured, the next step was to sort the data collected in a certain format to make the analysis of the results much easier. A small script was written using Flex to extract only the useful information which contained all the values of the different variables. The script is then converted to a C program which does the actual processing and outputs a new file which takes care of the formatting like spaces and separators in the proper format that could be used by another program that performs the analysis.

Data mining was performed on the captured and processed data to arrive at potential invariants. The code to perform this analysis was written in C++. The code contains 900 LOC. The fact that the number of relational operators is finite has been exploited to check the possibility of combinations of variables for satisfying a relationship. The program tries to explore all possible relations between pairs of variables. It checks for the following relations between 2 variables, say A and B:

***A > B***

***A >= B***

***A < B***

***A <= B***

***A == B***

***A != B***

The traces only provide data based on the conditions that were encountered during the execution of the program. However it does not say anything about the cases that were never executed or conditions that are not possible. To ensure that all possible paths have been executed, we need to assess the feasibility of other relations between the variables. The following were the potential invariants that were extracted from the data that was mined. These were the invariant relations that were not seen from the traces collected.

| | | |
|---|---|---|
| ***fin_angle*** | ***==*** | ***desired_pitch*** |
| ***fin_angle*** | ***>=*** | ***Actual_depth*** |
| ***fin_angle*** | ***==*** | ***Servo_cmd*** |
| ***fin_angle*** | ***>=*** | ***orientation_O*** |
| ***desired_pitch*** | ***==*** | ***Actual_depth*** |
| ***desired_pitch*** | ***>=*** | ***Servo_cmd*** |
| ***desired_pitch*** | ***==*** | ***orientation_O*** |

The negation of these potential invariants would serve as counterexamples since these were relations that are never seen. The next step was to verify if these potential invariants are true invariants are not. The verification of the true invariants can be done by a formal proof. In other case, if we could prove using a counterexample to show that the above can happen then we have verified that the invariant is false.

For this we have used a KLEE based tool called Cloud9 that can verify the relations using concolic execution techniques. The sections below further explain how this was achieved.

## 3.4 Tool decision

This section in brief talks about the reason for choosing cloud9 as the verification tool and not KLEE.

The test program was written in C++ with the use of Pthreads. KLEE has both self-contained package with binaries and also a development source code available. The self-contained package was used for the purpose of our verification. KLEE has limited support for C++ programs. Although there were libraries for POSIX runtime which linked along with the test program during the execution of KLEE and no errors were seen during conversion of the C++ code to object file or to byte code, there were warning generated when the following command was entered:

***klee --libc=uclibc --posix-runtime test.o***

The following were the warnings and errors generated.

***KLEE: NOTE: Using model:***
***/home/pgbovine/klee/Release+Asserts/lib/libkleeRuntimePOSIX.bca***
***KLEE: output directory = "klee-out-4"***
***KLEE: WARNING: undefined reference to function: klee_get_valuel***
***KLEE: WARNING: undefined reference to function: pthread_create***
***KLEE: WARNING: undefined reference to function: pthread_exit***
***KLEE: WARNING: undefined reference to function: pthread_join***
***KLEE: WARNING: executable has module level assembly (ignoring)***
***KLEE: WARNING: calling external: syscall(54, 0, 21505, 571522624)***
***KLEE: WARNING: calling __user_main with extra arguments.***
***KLEE: WARNING: calling external: pthread_create(571589384, 0, 563903904, 571574176)***
***0 klee 0x08965ab8***
***[pid 1846] +++ killed by SIGSEGV +++***
***+++ killed by SIGSEGV +++***
***Segmentation fault***

Using *llvm-ld* option during compilation to link pthread libraries did not help. It was realized that KLEE does not support pthreads. However there were other tools that were built on top of KLEE that extended support for pthreads. KLOVER for C++ and Cloud9 are some tools that extend this support.

KLOVER was not available for free and had very limited development support compared to Cloud9 that was available for development use for free. Hence the choice was made to use Cloud9 as the tool for verification of the code for AUV. Cloud9 has the ability to handle not only single-threaded programs but also multi-threaded and distributed systems. The symbolic environment model was the first to support all major aspects of the POSIX interface, such as processes, threads, synchronization, networking, IPC, and file I/O.

The section below explains in detail the steps for porting the program to Cloud9.

## 3.5 Porting to Cloud9 environment

Cloud9 works on 64-bit Linux distributions only. It has been tested on Ubuntu 10.04, 10.10, and 11.10, and it should work with other distros as well. Ubuntu 10.04 was the OS that was used. Cloud9 was installed according to the setup instructions provided on the website http://cloud9.epfl.ch/setting-up.

### 3.5.1 Error while building Cloud9

With the downloaded version of cloud9 the following error was encountered. This subsection provides details of the error and the ways to fix the error.

***sockets.c:1918:5: error: conflicting types for 'getnameinfo'***
***int getnameinfo(const struct sockaddr *sa, socklen_t salen,***
^
***/usr/include/netdb.h:679:12: note: previous declaration is here***
***extern int getnameinfo (__const struct sockaddr *__restrict __sa,***
^
***1 error generated.***

The error occurred because there was a mismatch between the header declaration of the ***getnameinfo()*** POSIX call in the OS, and the definition in the Cloud9 POSIX model.

The POSIX model is not fully independent of the host OS, as it references several system headers. The POSIX model had to be manually tweaked such that the declarations match the header definitions. The good news is that the headers usually differ only in minor ways (e.g., a parameter type is "int" instead of "long"), so the fix was straightforward.

This was fixed as below:

```
int getnameinfo(const struct sockaddr *sa, socklen_t salen,
        char *host, socklen_t hostlen,
        char *serv, socklen_t servlen,
        unsigned int flags)
```

The type for flags was changed to unsigned int from int to fix the error. The rest of the installation method follows the website above.

### 3.5.2 Creating the environment using gyp

Cloud9 uses Chromium's gyp [65] for building its internal LLVM bit code targets, and some other components. The make files are generated for the examples using gyp.

The input configurations are specified in files with the suffix .gyp. The .gyp file specifies how to build the targets for the "component" defined by that file. Each .gyp file generates one or more output files appropriate to the platform. On Linux, a .gyp file generates one SCons file and/or one Makefile per target. The .gyp file syntax is a Python data structure.

The .gyp file that was written contained the following.

```
{
  'targets': [
    {
      'target_name': 'depth',
      'type': 'executable',
      'dependencies': [
          '../build/libcxx.gyp:libcxx',
       ],
      'sources': [
        'depth.cpp',
      ],
      'include_dirs': [
        '../../cloud9/include',
      ],
```

```
      'link_settings': {
        'libraries': [
          '-pthread',
        ],
      },
    },
  ],
}
```

It specifies the target name, the dependencies, source files, include directories, and the link settings that set the library that it should be linked with it. More details of the gyp projects can be found at the website for gyp https://code.google.com/p/gyp/wiki/GypLanguageSpecification.

The default examples are placed inside *$CLOUD9_ROOT\src\testing_targets* directory. The test code was placed inside the *$CLOUD9_ROOT\src\testing_targets\myexample* directory. The newly added myexample folder and the .gyp file were included in the Makefile. On building the examples directory with the default Makefile using the command

***sudo build/make_all***

It builds all the examples along with the depth.cpp file.

## 3.6 Instrumentation of the source code

The source code for the depth controller only provides the actual functionality mentioned in the pseudo code. To verify the source code and to find invariants it is important to make the modifications in the code.

After making the changes for the building in the cloud9 environment, the code had to be instrumented with the constraints and assumptions that are needed for the tool to better understand the code.

The most critical step in using cloud9 is to inform the tool about the variables that should be executed symbolically and concretely. The tool has to be informed about other constraints in the code that are not explicit during the execution. This section explains the various instrumentations in the code that are required to use cloud9 for verification.

To use any of the KLEE utilities or functions, the header file for KLEE is to be included as below:

***#include <klee/klee.h>***

The following four KLEE functions were utilized in the instrumentation of the code:

- klee_make_symbolic
- klee_assume
- klee_thread_preempt
- klee_get_value_i32

The function declarations of these functions have to be added to the program that is tested as below:

```
void klee_make_symbolic(void *addr, size_t nbytes, const char *name) __attribute__((weak));
void klee_assume(uintptr_t condition) __attribute__((weak));
void klee_thread_preempt(int yield) __attribute__((weak));
extern "C"
{
int32_t klee_get_value_i32(int32_t) __attribute__((weak));
}
```

If these functions are declared in C++ code, it is important to make sure that the klee_get_value_i32() function declaration is wrapped in an extern "C" { } block. Otherwise, the compiler will mangle the names and the KLEE/Cloud9 runtime will not recognize them.

To verify and test a function or a program with KLEE/cloud9, it has to run on symbolic inputs. The *klee_make_symbolic()* function is used to mark a variable as symbolic. It takes three arguments: the address of the variable (memory location) that we want to treat as symbolic, its size, and a name (which can be anything). For example,

***klee_make_symbolic(&a, sizeof(a), "a");***

marks variable ‘a’ as symbolic.

Once the variables are made symbolic, cloud9 will execute this program symbolically on these variables and try to explore different states to catch bugs. This makes the input regular expression buffer completely symbolic that might try to explore lot of states that might not be relevant or might never be encountered in an AUV. It would be easier to achieve the required states if there exists a way to input the constraints that match the function of various functions. *klee_assume* intrinsic function provides a way to feed in such constraints to the tool.

*klee_assume()* takes a single argument (an unsigned integer) which generally should be some kind of conditional expression, and ”assumes” that expression to be true on the current path (if that can never happen, i.e. the expression is provably false, KLEE will report an error). *klee_assume* to cause KLEE to only explore states where the conditions are true. It can be used to encode more complicated constraints. For example,

***klee_assume(a != '0')***

causes KLEE to only explore states where the ‘a’ is not '0'. Multiple conditions can be clubbed together to form a much better constraint.

When executed symbolically on Cloud9, constraints get added for each instruction that is executed on the path. These symbolic expressions do not have concrete values for the variables until the expressions are sent to the solver. klee_get_value_i32() can be used to get satisfying

values for the variables in the symbolic expression by concretizing the values for the symbolic variables. For example, if x, a, b, d are symbolic variables

***a = x*100;***

***b =a+10;***

***d = a*b;***

***c = klee_get_value_i32(d);***

would assign ‘c’ a value that satisfies (x*100) * ( (x * 100) + 10).

Cloud9 by default follows cooperative scheduling. An enabled thread runs uninterrupted (atomically), until either of the following conditions is satisfied:

- The thread goes to sleep.
- The thread is preempted explicitly.
- The thread is terminated via symbolic system calls for process/thread termination.

The default Cloud9 scheduler switches threads only when a synchronization operation (e.g., waiting for a lock) is executed in the current thread. By changing the scheduling behavior of the symbolic execution engine, it is possible to preempt the current thread. *klee_thread_preempt()* can be used for preempting current thread. The section below explains how these functions were used in verifying the source code

## 3.7 Verifying the source code

As the first step a function had to be added which can contain all checks for the potential invariants that were newly deduced from the traces collected by simulating the source code. This function would have to be called from a separate monitor thread from the main as below:

```
pthread_create(&Monit_thread, NULL, run_Monitor_thread, NULL);

void* run_Monitor_thread(void *arg)
{
   Monitor_thread();
}
```

The objects of the different classes, which have functions run in parallel, were marked as symbolic to make them execute symbolically. Only those objects that are concrete were made as symbolic using the *klee_make_symbolic()*. Otherwise this would cause invalid make_symbolic error. This is done right after the objects of the class are created. The objects were made symbolic using the *klee_make_symbolic()* function as shown below:

```
void* run_calc_pitch(void *arg)
{
        Depth_Controller *Dep_Con = (Depth_Controller*)arg;
        klee_make_symbolic(&Dep_Con, sizeof(Dep_Con), "Dep_Con");
        Dep_Con->calc_pitch();
}
```

Similarly the objects for the other classes like the Orientation_Sensor, Depth_Sensor, Pitch_Controller, Fin_Actuator, and Depth_Monitor were also made symbolic.

The ranges for the various parameters were encoded using klee_assume. This makes sure that only the states that satisfy the ranges of different parameters are explored and asserts only with the right set of values of the AUV depth controller. The constraints were added after the set of statements that mark the inputs as symbolic, so that they are assumed from the beginning of the execution tree. The set of statements added to the test code can be seen below.

```
klee_assume(
        (Dep_Con->actual_depth > 0 & Dep_Con->actual_depth <= MAX_DEPTH) &
        (Dep_Con->depth_error >= 0 && Dep_Con->depth_error <= 2*MAX_DEPTH) &
        Dep_Con->depth_error_last >= 0 && Dep_Con->depth_error_last <= 2*MAX_DEPTH)
);
```

The '&' and '|' operators were used instead of the short-circuiting '&&' and '||' since with the Boolean operators KLEE branches the process before computing the result of the expression. This may result in exploring unnecessary additional states. Hence bit wise operators were used to reduce the number of states explored.

Inside the threads, the *klee_get_value_i32()* function was used to assign a concrete value for the symbolic expressions created in the path to the global copy of the different parameters like below:

```
glob_desired_pitch = klee_get_value_i32(desired_pitch);
```

The global value is then checked for invariants inside the monitor function. This monitor thread would be the last one to be created after all the other threads that collect and monitor data are created. Assert statements were inserted after every check added for the potential invariants. Cloud9 will try to cause these asserts to fail wherever possible. The values for the different parameters would be recorded into a file using fprintf whenever the conditions are met. Recording the values using fprintf is optional since KLEE automatically generates a test case whenever it branches into a condition. However it can be used to obtain the values of variables that are concrete since the test cases are generated only with the values of symbolic variables. The code below shows the check added for the potential invariant, which checks if the fin angle and desired pitch can be equal.

```
void Monitor_thread(void)
{
        pthread_mutex_lock( &glob_fin_angle_mutex );
        pthread_mutex_lock( &glob_desired_pitch_mutex );

        if(glob_fin_angle       ==       glob_desired_pitch)
        {
                outFile = fopen("out1.txt", "a");
                if (!outFile)
                {
                        fprintf(stderr, "Can't open out.txt\n");
                        exit(-1);
                }

                fprintf(outFile,"Fin_angle == Pitch \n");
                fprintf(outFile,"Servo_cmd = %d\n",glob_Servo_cmd);
                fprintf(outFile,"orientation_O = %d\n",glob_orientation);
                fprintf(outFile,"Actual_depth = %d\n",glob_actual_depth);
                fprintf(outFile,"fin_angle = %d\n",glob_fin_angle);
                fprintf(outFile,"desired_pitch = %d\n",glob_desired_pitch);
                fprintf(outFile,"\n\n");
                fclose(outFile);

                error_flag = 1;
        }

        assert(glob_fin_angle == glob_desired_pitch);

        pthread_mutex_unlock( &glob_desired_pitch_mutex );
        pthread_mutex_unlock( &glob_fin_angle_mutex );
…}
```

Since Cloud9 by default follows cooperative scheduling, to tackle the resource starvation problem caused by the Cloud9 scheduler, which let only one thread to run indefinitely not letting other threads to execute, *klee_thread_preempt()* calls were inserted at the end of every thread to call the next thread that reads the data that was written to in the current thread. This technique of preemption works fine in the AUV depth controller case since there is a clear pattern of publisher-subscriber mechanism and the variable is written only once in the entire code. The final thread after updating the variables preempts it, and passes the control to the monitor thread that performs the checking of the conditions for the potential invariants. This way the monitor thread is executed once after all other threads are executed once.

After the above changes were made, the modified code was built with the command mentioned in the previous section to create the byte code (depth.bc) of the original C++ file (depth.cpp). The command

***export GLOG_logtostderr = 1***

configures Cloud9 logging system to print the testing progress in the terminal, instead of writing it in a file under /tmp.

To spawn a Cloud9 worker that runs stand alone and uses the POSIX environment model to support all the API calls in the code, the following command was executed from the directory *$CLOUD9_ROOT\src\testing_targets*:

***$CLOUD9_ROOT/src/cloud9/Release+Asserts/bin/c9-worker --stand-alone --posix-runtime --libc=uclibc -use-forked-stp -fork-on-schedule --output-dir=test-depth out/Default/depth.bc --unsafe --no-overlapped***

By default, Cloud9 disallows writing into concrete files, in order to avoid accidental interference when multiple states write to the same file, independent of each other. The "--unsafe" option is passed to the target program argument list to allow the fprintf option in the code. Without this option Cloud9 does not allow writing to a file. To further control how overlapping is handled, the "--no-overlapped" option is passed in addition to "--unsafe". When this option is enabled, write requests that are originally issued to the same offset in the file are actually performed

sequentially in the file. This is useful, for instance, when the program opens a log file and you want to capture the log entries originating from all states.

Executing cloud9 with the command above, produces *.ktest file for different paths and whenever it encounters any errors. The file has information of the values of the symbolic variables. Along with the *.ktest file other files that are relevant to the errors in the code might also be produced. It produces an *.<error-type>.err, which is generated for paths wherever KLEE found an error. It contains the information of the error in a textual format. It can detect the following errors:

- **ptr:** Stores or loads of invalid memory locations.
- **free:** Double or invalid free().
- **abort:** The program called abort().
- **assert:** An assertion failed.
- **div:** A division or modulus by zero was detected.
- **user:** There is a problem with the input (invalid klee intrinsic calls) or the way KLEE is being used.
- **exec:** There was a problem which prevented KLEE from executing the program; for example an unknown instruction, a call to an invalid function pointer, or inline assembly.
- **model:** KLEE was unable to keep full precision and is only exploring parts of the program state.

To open a *.ktest file the ktest-tool which is provided with the KLEE is used in the following format:

***$CLOUD9_ROOT/src/cloud9/tools/ktest-tool/ktest-tool --write-ints test-depth/test000001.ktest***

More files can be generated for detailed information on the constraints or the queries can also be generated in different formats using various other command line options that can be supplied while invoking cloud9. Apart from file generation, there are other command line options that are available that can be used to change the search heuristics, max execution time, memory consumption and forks for symbolic branches.

Although the test cases generated by KLEE can be run by hand, KLEE provides a replay library, which can be used to replace the klee_make_symbolic with a call to a function that assigns the value stored in the .ktest file to the inputs. This can be done by linking the program with the libkleeRuntest library.

## 3.8 Results

On verifying the modified source code of the depth controller using Cloud9, it was seen that most of the conditions that were extracted as potential invariants are false. It was found that test cases were always generated such that it matches the conditions supplied to the tool using the assume statements. The values also matched the ranges of values for the parameters that were provided along with the pseudo-code for the depth controller.

The only invariant that was true was *fin_angle == Servo_cmd* is not possible, which is also very evident from the formula for calculating servo command which is

$$\textbf{\textit{Servo command } } S = K_{servo} * \delta_{fin}$$

This can never happen for non-zero value of fin angle and also the range of servo command should always lie between 1000 and 2000. With the range of the fin angle between -18 and +18 and servo constant $K_{servo}$ as 9.63, this invariant is never possible in a real scenario.

Another invariant that caused assertion failure was *desired_pitch >= Servo_cmd*. This is pretty obvious from the range of working for these parameters of the AUV that specifies that the pitch range between -25 to +25 and the servo command range between 1000 and 2000.

This condition can however be used as an invariant to detect malfunctioning of some sensors or to notify the system that there is some unusual situation which might require aborting the system.

These invariant can be used in a monitor thread to assert for the condition. In case of assert failure, suitable measures can be taken by the software.

The results were discussed with the Aeronautical team and it was confirmed that these potential invariants are false except for the ones mentioned above. The reason for these invariants to be false was specific to the operation of the AUV. The AUV can take any value for the parameters, without following a strictly increasing or decreasing curve, depending on the current orientation of the vehicle and change its values to guide itself to the right path with the right orientation. Hence these values might drastically from a negative value to a positive value and vice versa. The argument is also justified since these parameters are independent of each other and get their values from independent sources or sensors.

The analysis done using the above technique clearly proves that Cloud9 can be used as a verification tool to extract invariants from the code. Apart from extracting new invariants, it can also help in identifying other bugs that the tool is capable of uncovering like the ones mentioned in the previous section. This can help in avoiding and identifying human errors introduced into the system well before the system is deployed in the field. The tool also helps in generating other test cases that can make the test suite stronger than the manually written test cases. This verification technique when used with other tests can ensure a more reliable system.

The program for the depth controller that was written was only an approximate simulation of the actual depth controller software of an AUV. More invariants can possibly be extracted by using the above method when used on the actual software developed by the AUV team.

The next chapter explains in detail on how to extract invariants and verify the code using VCC.

# 4. Verification using VCC

This chapter explains in detail the program that was verified using VCC: Verifier for Concurrent C, the various steps involved in extracting invariants for the program and how they were analyzed and steps that were followed to verify the code using VCC.

Since VCC does not support C++ code, the input for VCC should be a C program. The section below explains how the pseudo-code was converted to C.

## 4.1 Simulating the C program

The input pseudo-code had to be implemented in C. The C program followed the same pattern as the C++ code, with an initialization function that initialized all the variables that were earlier initialized by the constructors in the various classes of the C++ code. The code used the POSIX thread to implement the multi-threaded model of the pseudo-code and ran to about 700 lines.

The various parallel processes of the pseudo-code correspond to a separate thread, which follows the same pattern as the C++ code above. Being a multi-threaded environment the possibilities of errors remains the same in both the C++ and C versions of the code. The program tries to eliminate some common problems of multi-threading by the use of mutex to avoid simultaneous read and write to the same variable.

The exit conditions in the C code remain the same as the one in C++ code. All threads exit whenever there is any violation seen by the Depth monitor thread which checks if the current depth is greater than the maximum depth and if the depth has not changed over a period of time.

The next step after conversion of the code to C was to identify the parameters of interest from the code and proceed towards extracting the invariants from the code.

## 4.2 Invariant extraction

The extraction of invariants is the next step after the code has been rewritten in C. The procedure for invariant extraction is the same as the invariant extraction explained in Section 3.3. The following are the parameters identified:

- desired pitch
- fin angle
- orientation
- depth
- servo command

The parameters remain the same as above since the functionality of the code has not changed.

As discussed previously, the code was instrumented to capture the traces of these parameters (variables) and the program was executed to collect the traces and capture the values of the variables. Scripts to extract the data from the captured traces were executed and the data mining was performed on the data to arrive at the potential invariants. The following were the potential invariants that were extracted from the data that was mined.

***fin_angle*** == ***desired_pitch***
***fin_angle*** == ***Actual_depth***
***fin_angle*** == ***Servo_cmd***
***fin_angle*** == ***orientation_O***
***desired_pitch*** >= ***Actual_depth***

***desired_pitch == Servo_cmd***

***desired_pitch >= orientation_O***

***Actual_depth == Servo_cmd***

***Servo_cmd == orientation_O***

These were the invariant relations that were not seen from the traces collected. The negation of these could be the potential invariants since these are relations that were never seen. The next step was to verify if these potential invariants are true invariants are not. The verification of the true invariants can be done by a formal proof. If not, if we could prove using a counterexample to show that the above can happen then we have verified that the invariant is false.

## 4.3 VCC build environment

The VCC was downloaded from the website http://vcc.codeplex.com/releases/view/101889. The version that was used was the Latest build, v2.3.10214.0, released on Feb 14, 2013. It is a windows installer and had to be installed in the default directory. The following prerequisites are needed to successfully run VCC:

- .Net v4.0 or later
- F# 2.0 Redistributable
- F# Power Pack
- Microsoft Visual C++ 2010 Redistributable Package
- Visual Studio 2010 or 2012 (any edition with C++ support). Note:However, the express editions are not recommended, because they will not allow the use of VCC from the IDE.

The software was installed on Windows7 Home premium on Intel Core i5 CPU at 2.30GHz and 8GB ram. Visual Studio 2010 was used and VCC was installed as an add-on-plugin to Visual Studio 2010.

VCC can be run from both command line and also from the Visual Studio IDE. VCC headers were added to the include paths if VCC is run from the IDE. The instructions to add the VCC headers to include path in the IDE are explained in the wiki page http://vcc.codeplex.com/wikipage?title=Install&referringTitle=Home.

A new project was created in Visual Studio 2010 and the C code for the depth controller was added to the project. Since VCC is integrated with Visual Studio the options for verifying in VCC are available on a right click. The IntelliSense in visual studio provides several VCC commands depending on the kind of construction of the C source file. The choice of verifying the entire file is always available. Clicking on the body of a function will provide VCC options for verifying that function alone.

Annotation of the code is the most critical step towards verification of the code using VCC. The following section explains in detail about the various steps involved in annotating the source code. It also highlights the various annotation tags and the types of annotation used during the modification of the code. It also discusses the approach adopted for the use of function contracts in the code and explains why this particular approach is adopted.

## 4.4 Annotating the source code

VCC uses modular verification technique. It performs a line-by-line checking of the source code. There are several advantages of this technique. Verification is made more scalable for larger programs. It allows the user to provide precise interfaces between the caller and the callee functions that make it easier to modify the implementation of a function without having to worry about breaking the verification of the function that calls it. This is advantageous because the person will have a chance to verify all the specification of the functions and would be able to recall any missed specification of a function. Also the calling function might not know about the

functionality of the called function in situations where the functions are implemented by two different sets of people. Another great advantage is that it is possible to verify a function even before the implementation is not available yet. The headers that are used for annotations can serve as the documentation of the function.

To verify that the program meets a certain set of specification, the inputs need to be fed to the tool to make it understand the purpose of the implementation. This set of inputs is fed by annotating the code with the tags.

Before annotating the code, a few changes were made to the existing program. Since the original code was written using pthreads, the pthread functions had to be removed. This was done since the final objective was to verify the individual functions, to verify the potential invariants and there were other means to specify to VCC that there are variables that could be modified by other threads in parallel. This could be done even without the functions being run in parallel. If it were possible to arrive at values that satisfy the constraints of the function and generate counter examples to falsify the potential invariants, it would prove that the invariants are false.

The mutexes in the code were removed for the same reason. Also since VCC supports modular verification with enough preconditions and post conditions it is possible to specify how the variable would be accessed and VCC would take care of the rest.

A new function was added as the monitor function, which would call the functions to be verified. In the case of the depth controller there were three functions (calculating pitch, fin angle and servo command) that performed the calculation and used the data from two other functions (supplying the depth and orientation). The three functions that performed the calculations were the ones that were targeted to be verified to check the invariants since they could be supplied with the necessary precondition and post conditions to interface with other functions.

The global variables which were earlier used in C code were converted into volatile global variables which notified VCC that these could be modified anywhere by other threads.

```
volatile int glob_fin_angle;
volatile int glob_actual_depth;
volatile int glob_orientation;
```

***volatile int glob_desired_pitch;***

***volatile int glob_Servo_cmd;***

The local variables inside the three functions to be verified (calc_pitch, calc_fin_angle and start_fin_actuator) were converted to global variables since there was a need to access these outside of the function to establish relationships between functions that updated and functions that accessed these variables.

***int desired_pitch=0, EI=0, ED=0, depth_error=0, actual_depth, depth_error_last = 0;***

***int orientation_P=0, pitch_error=0, EI_fin=0, orientation_P_rate=0, input_vec0=0, input_vec1=0, input_vec2=0, X=0, fin_angle=0;***

In addition to the above changes made to the C code, the program had to be included with annotations that are understandable only by the VCC tool. These annotations start with _(, terminating with a closing parenthesis, with balanced parentheses inside. The first identifier after the opening parenthesis is called an annotation tag and identifies the type of annotation provided. These annotations are used only by VCC, and are not seen by the C compiler. Before adding the annotations, the following header file was included:

***#include <vcc.h>***

When using the regular C compiler the <vcc.h> header file defines:

***#define _(...) /* nothing */***

VCC does not use this definition, and instead parses the inside of _(...) annotations.

### 4.4.1 Function annotations

The functions were first annotated with the annotation tags. The specification of a function is also called its contract, because it specifies the obligations on both the function and its callers. There are three main annotations that were used in the program to annotate functions:

- _(requires …)
- _(ensures …)
- _(writes …)

_(requires E), where E is an expression, is the precondition for the function and is a requirement on the caller. It specifies that the callers of the function would promise that E would hold on function entry. For example,

***_(requires \thread_local (&actual_depth) && actual_depth >= 0 && actual_depth < MAX_DEPTH)***
***_(requires \mutable (&glob_desired_pitch) && glob_desired_pitch >= -25 && glob_desired_pitch <= 25)***

*\thread_local* specifies that the object specified as thread_local is transitively owned by the current thread. To safely access memory sequentially, the memory must not be concurrently written by other threads (including hardware and devices). It is possible to read memory sequentially if it is a nonvolatile field of an object that is known to be closed, even if it is not owned by the thread; this allows multiple threads to sequentially access shared ROM.

*\mutable* specifies that the object specified as mutable is an open one owned by the current thread. To write sequentially through an object, it should be ensured that no other thread is trying to read (sequentially or concurrently) through the object at the same time. We write this as \mutable(object). Mutability makes sense only in the context of a particular thread.

_(ensures E), where E is an expression, is a post condition of the function and hence a requirement on the function. It specifies that the function that E would hold true just before control exits the function. For example,

***_(ensures depth_error == (desired_depth - actual_depth) )***

_(writes E) requires that the objects/fields pointed to by E are writable, and that a call to this function is allowed to change these fields/object. It specifies that a function needs writability of E at function entry. On a call to a function, VCC assumes that all of the objects listed in the writes clauses are writable on function entry. Moreover, if an object becomes mutable (for a thread) after entry to a function call, it is considered writable within that call (as long as it remains mutable). For example,

***_(writes &depth_error)***

### 4.4.2 Loop annotations

There are loops in almost all the functions of the depth controller. VCC allows the user to specify the invariants of the loop rather than allowing the tool to guess the conditions that should be satisfied at the top of a loop. To make sure that loop invariants indeed hold whenever control reaches the top of the loop, VCC asserts that the invariants hold wherever control jumps to the top of the loop - namely, on loop entry and at the end of the loop body. The formulae inside the depth controller processes were given as loop invariants apart from other conditions that were to be satisfied inside the loop. The loop invariants should be specified using the _(invariant …) annotation tag. For example,

***_(invariant fin_angle == C*X + (D[0]*input_vec0 + D[1]*input_vec1 +***
***D[2]*input_vec2) )***
***_(invariant depth_error >= 0 && depth_error <= 2*MAX_DEPTH)***

The code snippet below shows how the function contracts and loop invariants are specified for the function calc_pitch. It does not show the exhaustive list of annotations used of the function but gives a picture of how it was used.

```
void calc_pitch()
        _(requires \thread_local (&rate_f) &&  rate_f == RATE_F)
        _(requires \thread_local (&KP) &&  KP == CONTROLLER_GAIN_KP)
        _(requires \thread_local (&KI) &&  KI == CONTROLLER_GAIN_KI)
        _(requires \mutable (&glob_actual_depth) &&  glob_actual_depth >= 0 &&
glob_actual_depth < MAX_DEPTH)
        _(requires \thread_local (&actual_depth) &&  actual_depth >= 0 &&
actual_depth < MAX_DEPTH)
        _(requires depth_error  == (desired_depth - actual_depth) )
        _(requires desired_pitch == KP*depth_error + KI*EI+ KD*ED )
        _(writes &actual_depth)
        _(writes &depth_error)
        _(ensures rate_f == RATE_F)
        _(ensures KP == CONTROLLER_GAIN_KP)
```

```
        _(ensures KI == CONTROLLER_GAIN_KI)
        _(ensures glob_actual_depth >= 0 && glob_actual_depth <= MAX_DEPTH)
        _(ensures glob_desired_pitch >= -25 && glob_desired_pitch <= 25)
        _(ensures desired_pitch >= -25 && desired_pitch <= 25)
        ….
{
        int iteration = 0;
        while(!(exit_calc_pitch))
        _(invariant desired_pitch >= -25 && desired_pitch <= 25)
        _(invariant depth_error >= 0 && depth_error <= 2*MAX_DEPTH)
        _(invariant depth_error_last >= 0 && depth_error_last <= 2*MAX_DEPTH)
        _(invariant depth_error  == (desired_depth - actual_depth) )
        _(invariant desired_pitch == KP*depth_error + KI*EI+ KD*ED )
        _(invariant glob_desired_pitch == desired_pitch)
        _(invariant pitch_error == desired_pitch - orientation_P)
        {
                actual_depth = get_depth();
                depth_error  = _(unchecked)(desired_depth - actual_depth);
                …
                desired_pitch = _(unchecked)(KP*depth_error + KI*EI+ KD*ED);
                glob_desired_pitch = desired_pitch;
        }
}
```

These invariants can also be redundant, i.e., can also contain constraints that were originally part of the function contract. This only further enhances the verification by ensuring that the constraints hold true in places where it has to hold true and can be used to expose the errors in the code.

## 4.5 Verifying the annotated code

After adding the pre and post conditions for every function of the depth controller, it was possible to verify individual functions by right clicking the function on Visual studio and choosing 'verify this' option in the list of options. On verifying individual functions, the verification sometimes failed due to the possibility of overflow in some calculations. To indicate that this possible overflow behavior is desired we use _(unchecked), with a syntax similar to a regular C type-cast. This annotation applies to the following expression, and indicates that you expect that there might be overflows in there. _(unchecked)E could be imagined as computing the expression using mathematical integers, which never overflow, and then casting the result to the desired range. For example,

***EI = _(unchecked)(EI + depth_error/rate_f);***

There were possibilities of divide by zero which caused verification failures. These situations were tackled by using _(assume) annotations in places where it was known that zero values for those variables were not possible. _(assume E), where E is an expression, informs VCC to ignore the rest of an execution if E fails to hold (or, more operationally) wait until E holds. For example, the following statement in the code caused a verification failure due to possibility of a division by zero with the variable rate_f.

***EI = _(unchecked)(EI + depth_error/rate_f);***

This was tackled by using the following annotation tag before the statement that makes sure that it will assume a non-zero value for the variable when it encounters the statement with the division.

***_(assume rate_f != 0)***

The helper get-functions that read the values from the global variables in the original depth controller code were also annotated with _(assume) tags to return values in the range mentioned in the specification for the pseudo-code for the depth controller. For example,

```
int get_Fin_angle(void)
_(requires \mutable(&glob_fin_angle))
_(ensures glob_fin_angle >= -18 && glob_fin_angle <= 18)
{
        _(assume glob_fin_angle >= -18 && glob_fin_angle <= 18)
        return(glob_fin_angle);
}
```

After every function was verified individually, a new monitor function was written which called functions that were to be verified. The monitor function was annotated with preconditions and post conditions that not only individually verified the calls to the functions but also had constraints that interfaced the relationships between the called functions. While interfacing the functions, there were additional constraints that had to be added to the individual function. These explain the reason for making some local variables in the original unmodified C code variables into global during verification.. For example,

```
_(requires pitch_error == desired_pitch - orientation_P)
_(ensures pitch_error == desired_pitch - orientation_P)
```

were constraints added in the calc_pitch function, although there was no calculations for pitch_error inside the calc_pitch function.

After the constraints to interface, the called functions were added to the monitor function and the called functions inside it. The next step was to verify the feasibility of the potential invariants extracted with the traces. This was done using the _(assert …) annotation tag. _(assert E), where E is an expression, which tries to prove that E holds at that point, and assumes E afterward. For example,

```
_(assert glob_desired_pitch      <     glob_orientation)
```

The _(assert E) that VCC uses is quite different from the macro assert(E) defined in the header file <assert.h>. The latter evaluates E at runtime and aborts the execution if E does not hold. It involves overhead in builds where the check is made. It catches assertion failures only when it actually fails during the execution of the program. _(assert E) overcomes these problems since it

would guarantee to catch the assertion failure if it is possible in any execution. Since _(assert E) is not actually executed, E can include mathematical operations such as quantification over infinite domains that are not implementable.

After adding annotation tags and assert statements the monitor function looks like the below:

```
void Monitor_thread()
_(requires \thread_local (&desired_depth) &&  desired_depth == MAX_DEPTH)
_(requires \thread_local (&rate_f) &&  rate_f == RATE_F)
_(requires \thread_local (&KP) &&  KP == CONTROLLER_GAIN_KP)
_(requires \thread_local (&KI) &&  KI == CONTROLLER_GAIN_KI)
_(requires \thread_local (&KD) &&  KD == CONTROLLER_GAIN_KD)
_(requires \thread_local (&A) &&  A == CONTROLLER_MATRIX_A)
_(requires \thread_local (&B[0]) &&  B[0] == CONTROLLER_MATRIX_B_0)
_(requires \thread_local (&B[1]) &&  B[1] == CONTROLLER_MATRIX_B_1)
_(requires \thread_local (&B[2]) &&  B[2] == CONTROLLER_MATRIX_B_2)
_(requires \thread_local (&C) &&  C == CONTROLLER_MATRIX_C)
_(requires \thread_local (&D[0]) &&  D[0] == CONTROLLER_MATRIX_D_0)
_(requires \thread_local (&D[1]) &&  D[1] == CONTROLLER_MATRIX_D_1)
_(requires \thread_local (&D[2]) &&  D[2] == CONTROLLER_MATRIX_D_2)
_(requires \mutable (&glob_actual_depth) &&  glob_actual_depth >= 0 &&
glob_actual_depth < MAX_DEPTH)
_(requires \thread_local (&actual_depth) &&  actual_depth >= 0 && actual_depth <
MAX_DEPTH)
_(requires \mutable (&glob_desired_pitch) && glob_desired_pitch >= -25 &&
glob_desired_pitch <= 25)
_(requires \thread_local (&desired_pitch) && desired_pitch >= -25 && desired_pitch
<= 25)
_(requires \thread_local (&depth_error) &&  depth_error >= 0 && depth_error <=
2*MAX_DEPTH)
_(requires \thread_local (&depth_error_last) && depth_error_last >= 0 &&
depth_error_last <= 2*MAX_DEPTH)
```

*_(requires depth_error  == (desired_depth - actual_depth) )*

*_(requires desired_pitch == KP*depth_error + KI*EI+ KD*ED )*

*_(requires glob_desired_pitch == desired_pitch)*

*_(requires \mutable (&glob_orientation) && glob_orientation >= -15 && glob_orientation <= 15)*

*_(requires \thread_local (&orientation_P) &&  orientation_P >= -15 && orientation_P <= 15 )*

*_(requires \thread_local (&orientation_P_rate)  && orientation_P_rate >= -10 && orientation_P_rate <= 10 && orientation_P_rate != 0)*

*_(requires \mutable (&glob_fin_angle) && glob_fin_angle >= -18 && glob_fin_angle <= 18)*

*_(requires \thread_local (&fin_angle) && fin_angle >= -18 && fin_angle <= 18)*

*_(requires \thread_local (&pitch_error) &&  pitch_error >= -40 && pitch_error <= 40 )*

*_(requires pitch_error  == desired_pitch - orientation_P)*

*_(requires input_vec0 == orientation_P_rate)*

*_(requires input_vec1 == pitch_error)*

*_(requires input_vec2 == EI_fin)*

*_(requires X == A*X + (B[0]*input_vec0 + B[1]*input_vec0 + B[2]*input_vec2) )*

*_(requires fin_angle == C*X + (D[0]*input_vec0 + D[1]*input_vec1 + D[2]*input_vec2) )*

*_(requires glob_fin_angle == fin_angle)*

*_(writes &glob_desired_pitch)*

*_(writes &actual_depth)*

*_(writes &depth_error)*

*_(writes &EI)*

*_(writes &ED)*

*_(writes &depth_error_last)*

*_(writes &desired_pitch)*

*_(writes &glob_fin_angle)*

*_(writes &orientation_P)*

```
_(writes &orientation_P_rate)
_(writes &pitch_error)
_(writes &EI_fin)
_(writes &input_vec0)
_(writes &input_vec1)
_(writes &input_vec2)
_(writes &X)
_(writes &fin_angle)
_(ensures glob_actual_depth >= 0 && glob_actual_depth <= MAX_DEPTH)
_(ensures actual_depth >= 0 && actual_depth < MAX_DEPTH)
_(ensures glob_desired_pitch >= -25 && glob_desired_pitch <= 25)
_(ensures desired_pitch >= -25 && desired_pitch <= 25)
_(ensures depth_error >= 0 && depth_error <= 2*MAX_DEPTH)
_(ensures depth_error_last >= 0 && depth_error_last <= 2*MAX_DEPTH)
_(ensures depth_error  == (desired_depth - actual_depth) )
_(ensures desired_pitch == KP*depth_error + KI*EI+ KD*ED)
_(ensures glob_desired_pitch == desired_pitch)
_(ensures glob_orientation >= -15 && glob_orientation <= 15)
_(ensures  orientation_P >= -15 && orientation_P <= 15 )
_(ensures orientation_P_rate >= -10 && orientation_P_rate <= 10 &&
orientation_P_rate != 0)
_(ensures glob_fin_angle >= -18 && glob_fin_angle <= 18)
_(ensures fin_angle >= -18 && fin_angle <= 18)
_(ensures  pitch_error >= -40 && pitch_error <= 40 )
_(ensures pitch_error  == desired_pitch - orientation_P)
_(ensures input_vec0 == orientation_P_rate)
_(ensures input_vec1 == pitch_error)
_(ensures input_vec2 == EI_fin)
_(ensures X == A*X + (B[0]*input_vec0 + B[1]*input_vec0 + B[2]*input_vec2) )
_(ensures fin_angle == C*X + (D[0]*input_vec0 + D[1]*input_vec1 +
D[2]*input_vec2) )
```

```
_(ensures glob_fin_angle == fin_angle)
{
        calc_pitch();
        calc_fin_angle();
        _(assert glob_desired_pitch     <     glob_orientation)
}
```

To verify the program and the feasibility of the potential invariants, right click on the code in Visual Studio and click on ‘verify file’. Annotated programs are translated to logical formulas using the Boogie tool. VCC uses the deductive verification paradigm: it generates a number of mathematical statements (called verification conditions). The validity of these conditions is enough to guarantee the program's correctness. These conditions are passed to an automated theorem prover (Z3) to check their validity.

When VCC is run, several outcomes are possible: VCC could report that the program is correct, in which the program is guaranteed to satisfy all of the annotations. VCC could report that it is unable to verify the correctness of one or more of the annotations, in which case one can use the VCC Model Viewer to inspect how VCC thinks the program (or the description of why it works) might fail. The Model Viewer is available as a GUI that is bvd.exe. The .model file, which is generated in case of verification failures, can be loaded into the GUI to see the values assigned to the variables that form the mathematical formulae. Figure 4.1 shows a screenshot of the GUI with .model file opened.

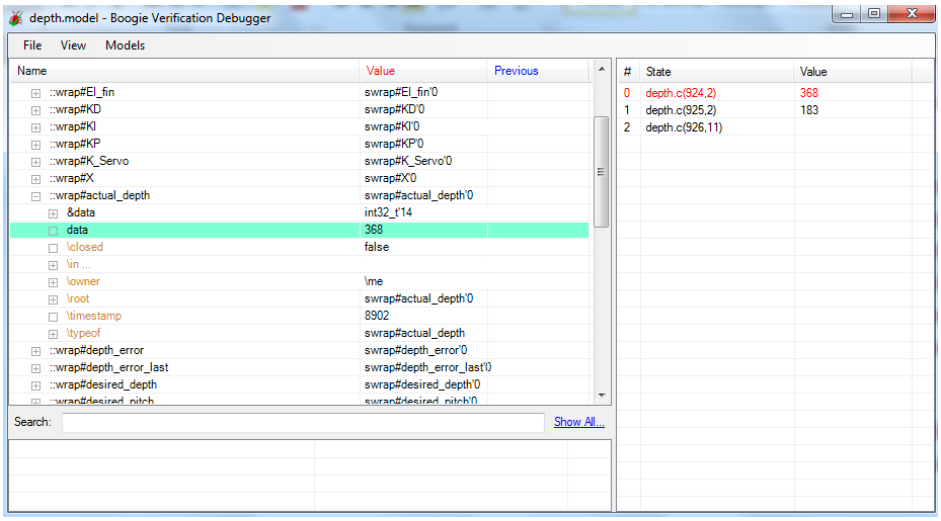


**Figure 4.1 VCC model viewer**

VCC Inspector can be used to monitor the proof progress to see which part of the program is causing the verifier to get stuck or the Z3 Axiom Profiler to see where the prover is spending its time in cases where the theorem prover might diverge.

## 4.6 Results

On verifying the source code for depth controller using VCC, it was seen that the conditions that were extracted as potential invariants are false. VCC was able to generate values for the constraints that were supplied to the tool as preconditions and values generated satisfied the post conditions. The values also matched the ranges of values for the parameters that were provided along with the pseudo-code for the depth controller.

The only invariant that was true was *fin_angle == Servo_cmd* is not possible, which is also very evident from the formula for calculating servo command, which is

$$\textbf{\textit{Servo command } S = } K_{servo} * \delta_{fin}$$

This can never happen for non-zero values of fin angle and also the range of servo command should always lie between 1000 and 2000. With the range of the fin angle between -18 and 18 and servo constant $K_{servo}$ at 9.63, this invariant is never possible in a real scenario. This condition can however be used as an invariant to detect malfunctioning of some sensors or to notify the system that there is some unusual situation which might require aborting the system. The invariant can be used in a monitor thread to assert for the condition fin angle is not equal to the servo command. In case of assert failure, suitable measures can be taken by the software.

The results were discussed with the Aeronautical team and it was confirmed that these potential invariants are false except for the one mentioned above. It was also discussed that any of these variables could take any value in the specified range anytime depending on the orientation and the physical condition of the Autonomous vehicle and they do not have to necessarily follow a strictly increasing or decreasing pattern. The vehicle might have to vary a particular parameter from a negative value to a positive value or vice versa to orient itself in the proper direction. As long as the values satisfy the formula the values that it can take are completely acceptable. This conclusion also seems plausible because the variables are mostly independent of each other having their source of values obtained from different set of sensors.

The analysis done using the above technique clearly proves that VCC can be used as a verification tool to extract invariants from the code. Apart from extracting new invariants, it also aids in uncovering other bugs in the software that might be caused due to human errors. The program for the depth controller that was written was only an approximate simulation of the actual depth controller software of an AUV. More invariants can possibly be extracted by using the above method when used on the actual software developed by the AUV team.

VCC performs a modular verification. By exploiting the advantages of this, the developer can use the tool well before the complete software is ready to be deployed in the field and take necessary safety measures in the software to tackle error situations, thus reducing the cost of the errors and reducing the human interactions and making the AUV more self-monitoring. The constraints specified to VCC, apart from guaranteeing verified software, can also be part of the software documentation plan, which is required towards certification for the AUV. The next chapter concludes the thesis work and provides recommendations and directions for future work.

# 5. Conclusion and recommendations

This chapter concludes the work done as part of the thesis providing the justification of how the work can be useful in increasing the robustness of the system and provides recommendations for the AUV software team to improve the quality of the system and provides directions for future work.

## 5.1 Conclusion

The thesis begins with the discussion about the influence of software in various sectors and how software systems are becoming more universal in diverse applications, including cyber-physical systems like Autonomous Underwater Vehicles. It explains the impact of a failure in complex systems and how reliability of the system increases with more robust testing and verification. It identifies verification as the most important phase of the Software Development Life Cycle to reduce the cost to the industry arising due to development errors and bugs identified post-release.

It gives a holistic perspective of how stringent the requirements of safety critical software are, taking unmanned vehicles in general as an example. The requirements for certification of the software such systems and the software development process for safety software, are highlighted. It enlists the various software metrics that are used to measure the size and complexity of the software. The pattern correlating to the annual increase in the size and complexity of code is shown. It emphasizes the need for effective verification as a means to guarantee reliability of software.

As a means to verify the software for Autonomous Underwater Vehicles (AUV), we have used the technique of extracting potential invariants from the source code by producing traces of the variables from the source code and used two tools namely - Cloud9 and VCC. This enabled us to eliminate the false invariants by generating test cases that serve as counter examples to disprove the invariants with valid values that conform to the constraints for each variable. The experiments conducted prove that these tools namely, Cloud9 and VCC can be used along with invariant based methods to explore new invariants. The new invariants identified can be used to as monitors to detect malfunctioning of the AUV or alert the system for any erroneous situation.

Apart from verifying and identifying new invariants, this method also generates new test cases that can be used by the testing team to test all possible paths in the program or test individual functions. These test cases can be a part of the test suite along with other manual test cases that can make guarantee better robustness of the system. The method also helps in uncovering and eliminating other human introduced bugs in the software - like pointer errors, Stores or loads of invalid memory locations, division or modulus by zero errors or overflows that might have been missed by mere testing.

The strategy identified new invariants for the AUV by exploring all the paths in the code leading to a better branch coverage and assertion coverage. This method ensures safety and correctness of the AUV software and the system behavior, increases the reliability and reduces the cost of in-field testing by eliminating the risk of human errors well before it could be deployed in the field.

This method of identifying new invariants can also be used in other embedded systems that use software which require publishing of information to other modules and subscribing to the information from other modules.

## 5.2 Recommendations for future work

This section suggests recommendations to the AUV team that can be incorporated in their software to improve the quality of the entire system.

The work addressed in the thesis so far assumes that there are no hardware failures in the system. To detect malfunctioning of the hardware, apart from the default self-test of the hardware, there can be other threads to monitor the parameters for the trend of the variation of these different parameters for a period of time. If there are changes in the trend too frequently it can be notified. For example, if the speed sensor indicates 5 miles per hour, but the depth is increasing at greater than 5 miles per hour, then it would be evident that at least one of the two sensors might be defective. A thread can be used to abort the AUV due to erratic behavior of the variations. This could mean that there is some harsh condition underwater. But if this behavior is seen very persistently then this could mean some critical error. By using other physical parameters or using other related sensor data, given a means to determine the actual intended behavior of the AUV, the current behavior can be matched with the actual intended behavior to foresee failure of hardware or in the worst case abort the system to prevent any other hazardous activity. This thread can be used to catch errors in situations where the AUV is expected to move deeper but it is not actually moving in the right path as intended. The example thread below shows how this can probably be implemented.

```
void monitor_variation_trends()
{
        int depth_0,depth_1, orientation_0, orientation_1;
        int error_flag, count;
        int delta_depth,delta_orientation;
        int depth_increasing,orientation_increasing;
        int prev_depth_trend,depth_retry_count;
        int prev_orientation_trend,orientation_retry_count;
        error_flag = count  = 0;
```

```
prev_depth_trend = depth_increasing = 0;
depth_retry_count = orientation_retry_count = 0;
prev_orientation_trend = orientation_increasing = 0;
depth_0 = get_depth();
orientation_0  = get_orientation();
while(!error_flag){
        while(count <= 1000) {
                count++;
        }
        depth_1 = get_depth();
        orientation_1 = get_orientation();
        delta_depth = depth_1 - depth_0;
        delta_orientation = orientation_1 - orientation_0;
        if(delta_depth < 0) {
                depth_increasing = -1;
        }else if (delta_depth > 0) {
                depth_increasing = 1;
        }else{
                depth_increasing = 0;
        }
        if(prev_depth_trend != depth_increasing) {
                depth_retry_count++;
                if(depth_retry_count >= 10) {
                        // error depth trend varying very frequently
                        abort();
                }
        }else{
                depth_retry_count = 0;
        }
        prev_depth_trend = depth_increasing;
```

```
                    if(delta_orientation < 0) {
                            orientation_increasing = -1;
                    }else if (delta_orientation > 0) {
                            orientation_increasing = 1;
                    }else{
                            orientation_increasing = 0;
                    }
                    if(prev_orientation_trend != orientation_increasing) {
                            orientation_retry_count++;
                            if(orientation_retry_count >= 10) {
                                    // error orientation trend varying very frequently
                                    abort();
                            }
                    }else{
                            orientation_retry_count = 0;
                    }
                    prev_orientation_trend = orientation_increasing;
                    depth_0 = get_depth();
                    orientation_0 = get_orientation();
            }
    }
```

The thread can be further extended to observe trend variations between two or more parameters. For example, assume that an increasing depth trend and decreasing orientation trend is initially observed and at a later time one of these change the variation pattern - like decreasing depth trend and decreasing orientation trend (which is possible in a real scenario) – then this is acceptable. If the variations between these two parameters keep changing very frequently this might imply that something is wrong. Such a monitor thread can be really helpful in a scenario like this and can be included along with other threads that monitor the system. More invariant relations can be uncovered using such threads. However this thread should also be verified using some formal methods for verifying the code.

The code can be instrumented to have line counters. A line counter keeps track of the number of times a line in the original code is executed. Consider the code snippet below,the numbers at the beginning of each line is the line number.

```
0  void function1(int F, int d) {
1        if (((F) % 20) != 0) {
2                a = d;
3                F = F + 1;
4                assert (function2(F) <= 20);
5         }
6  }
```

After code instrumentation, the code would look like below.

```
0  void function1(int F, int d) {
1        if (((F) % 20) != 0) {
2                a = d;
3                 line_encountered[2]++;
4                F = F + 1;
5                line_encountered[3]++;
6                assert (function2 (F) <= 20);
7                line_encountered[4]++;
8        }
9  }
```

The line_encountered is incremented every time the line is executed. The line_encoutered is a global array variable. Hence the value of the counter can be known at any point. The counter values can be updated in the EEPROM before the system aborts. This can be done on field. With this information, the condition from which the system aborted could be known, and this can help in post failure diagnosis.These recommendations can aid in improving the safety and reliability of the system and increase the quality of the final product and should be incorporated in the software for such complex and critical systems.

## 5.3 Future work

The work done with KLEE and cloud9 assumes all variables as integers since these tools do not support floating point. With the support for floating point, the technique of obtaining invariants and verification in general can be applied to systems with increased precision. There is a KLEE-related project on symbolically checking the equivalence between floating point values, but it doesn't allow general purpose symbolic floating-point computation [66].

Some solvers like Z3 support solving constraints with real numbers - that is, handle the precise version of floating point computations. For algorithms that do not rely on precision-related corner cases of floating point computations, it might be possible to obtain useful results by using reals instead of floats. This can however be implemented in Cloud9.

Implementation would involve integrating KLEE/Cloud9 with Z3 solver and investigating how and when floating point computations could be approximated by real computations. The constraints in KLEE or Cloud9 are not in the SMT-LIB format, but are in-memory data structures. KLEE keeps them in memory using special non-standard data structures. The STPBuilder conversion tool converts them into STP-specific data structures.

Z3 can be integrated with Cloud9 in the following ways:

1. Use Z3's API to programmatically construct expressions using Z3's own data structures.
2. Serialize the KLEE expressions into the SMT-LIB format and pass them to Z3.

KLEE supports serializing expressions into a special KQuery format, which can then be converted to SMT-LIB. Many of the queries are typically solved very fast and the time for them is already dominated by the translation/solver invocation step. Adding ASCII serialization/de-serialization into this step might increase the overhead. This has been tried already and the source is available at the location https://github.com/delcypher/klee/tree/smtlib. The results show that this is pretty slow.

A version of KLEE that supports Z3, Bootlector and STP has been recently developed using the metaSMT framework [67] which provides a unified API for using a number of SMT and SAT solvers with their own API's [68].

To integrate Z3 to cloud9 Z3 C++ api can be used to construct the expressions. The most important component required to integrate a new solver in Cloud9 (and KLEE, in general) is the expression builder class, which is responsible for converting a klee Expr object to an equivalent solver formula. STP's builder is in $CLOUD9_ROOT/src/cloud9/lib/Solver/STPBuilder.cpp. Secondly, a new Solver class needs to be created to implement the interface to the new solver. The StpSolverImpl class in $CLOUD9_ROOT/src/cloud9/lib/Solver/Solver.cpp shows how to do it for STP.

Integrating Cloud9 with Z3 solver would be a much simpler task. The future experiments and research should try to investigate how and when floating point computations could be approximated by real computations.

# Bibliography


[1] R. Wolfig, "Parameters for Efficient Software Certification," *itk.ntnu.no,* pp. 1-10.

[2] Paul Krill (InfoWorld), "Chasing bugs away," 08 December 2003. [Online]. Available: http://www.computerworld.com.au/article/119051/chasing_bugs_away/.

[3] Charette, RN, "This car runs on code," *IEEE Spectrum,* pp. 1-6, 2009.

[4] NASA Official: Brian Dunbar, "Facts and Figures," 28 July 2013. [Online]. Available: http://www.nasa.gov/mission_pages/station/main/onthestation/facts_and_figures.html.

[5] M. Dowson, "The Ariane 5 software failure," *ACM SIGSOFT Software Engineering Notes,* vol. 22, no. 2, p. 84, 1997.

[6] B. Nuseibeh, "Ariane 5: Who Dunnit?," *IEEE Software,* vol. 14, no. 3, pp. 15-16, 1997.

[7] M. Newman, "Software errors cost US economy $59.5 billion annually," *Department of Commerce's National Institute of Standards and Technology (NIST),* pp. 15-16, 2002.

[8] James G. Bellingham, University National Oceanographic Laboratory Systems., The Global Abyss: An Assessment of Deep Submergence Science in the United States, Deep Submergence Science Committee, 1994, p. 53.

[9] Massachusetts Institute of Technology, " Autonomous Underwater Vehicles," [Online]. Available: http://auvlab.mit.edu/research/AUVoverview.html.

[10] S. Mahmud, "REU PROGRAM IN TELEMATICS AND CYBER PHYSICAL SYSTEMS: SHARING STRATEGIES, EXPERIENCE AND LESSONS LEARNED TO HELP OTHERS," 2010.

[11] E. Lee, "The problem with threads," *Computer,* vol. 39, no. 5, pp. 33-42, 2006.

[12] N. Zeldovich, A. Yip, R. F. Dabek, T. Morris, D. Mazieres and F. Kaashoek, "Multiprocessor support for event-driven programs," *USENIX Annual Technical Conference, San Antonio, Texas, USA*, pp. 239-252, 2003.

[13] R. (. Rajkumar, I. Lee, L. Sha and J. Stankovic, "Cyber-physical systems," in *Proceedings of the 47th Design Automation Conference on - DAC '10*, 2010.

[14] J. Sztipanovits and G. Karsai, "Model-integrated computing," *Computer*, vol. 30, no. 4, pp. 110-111, 1997.

[15] Woods Hole Oceanographic Institution, "Autonomous Underwater Vehicles," Woods Hole Oceanographic Institution, [Online]. Available: http://www.whoi.edu/main/auvs.

[16] Bluefin Robotics, "Applications," Bluefin Robotics, [Online]. Available: http://www.bluefinrobotics.com/applications/.

[17] [Online]. Available: http://en.wikipedia.org/wiki/Invariant_(mathematics).

[18] [Online]. Available: http://en.wikipedia.org/wiki/Invariant_(computer_science).

[19] M. Ernst and J. Cockrell, "Dynamically discovering likely program invariants to support program evolution," *Software Engineering, IEEE Transactions*, vol. 27, no. 2, pp. 99-123, 2001.

[20] M. Ernst, "Dynamically discovering likely program invariants," 2000.

[21] M. DeGarmo, "Issues concerning integration of unmanned aerial vehicles in civil airspace," *The MITRE Corporation Center for Advanced Aviation System Development*, no. November, 2004.

[22] "Aviation Today," [Online]. Available: http://www.aviationtoday.com/regions/usa/2307.html#.Ud5HnUHVCSp.

[23] Camus, JL;Dion, B, "Efficient development of airborne software with Scade suite," *Esterel*

*Technologies,* pp. 1-49, 2003.

[24] Y. Lee and J. Krodel, "Flight-Critical Data Integrity Assurance for Ground-Based COTS Components," 2006.

[25] "GNAT Pro Safety-Critical," AdaCore, [Online]. Available: http://www.adacore.com/gnatpro-safety-critical/avionics/do178b/.

[26] Coverity Inc., "The Software Development Challenge".

[27] Federal Aviation Administration, "Commercial Off-The-Shelf (COTS) Avionics Software Study," DOT/FAA/AR-01/26, 2001.

[28] SAE – Systems Integration Requirements Task Group, " Certification Considerations for Highly-Integrated or Complex Aircraft Systems," ARP 4754, 1996.

[29] FAA, "System Software Safety," in *FAA System Safety Handbook*, 2000.

[30] L. Westfall, "12 Steps to Useful Software Metrics," *The Westfall Team, Plano.,* 2005 .

[31] S. Chidamber and C. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, p. 476–493, 1994.

[32] J. Bansiya and C. Davis, "A Hierarchical Model for Object-Oriented Design Quality Assessment," *IEEE Transactions on Software Engineering*, vol. 28, no. 1, pp. 4-17, 2002.

[33] M. Kozak, "Correlation coefficient and the fallacy of statistical hypothesis testing," *Current Science,* vol. 95, no. 9, 2008.

[34] David Shen, WCI, Inc;Zaizai Lu, AstraZeneca Pharmaceuticals, "Computation of Correlation Coefficient and Its Confidence Interval in SAS," 2006.

[35] A. Alhussein, H. Zedan, H. Janicke and O. Alshathry, "Software Certification through Quality Profiling," in *2009 International Conference on New Trends in Information and Service Science*, 2009.

[36] D. Dvorak, "NASA Study on Flight Software Complexity," in *AIAA Infotech@Aerospace Conference and AIAA Unmanned...Unlimited Conference*, 2009.

[37] N. Augustine, Augustine's Laws, 6th Edition ed., American Institute of Aeronautics and Astronautics., June 1977.

[38] S. D. Conte, H. E. Dunsmore and V. Y. Shen, Software engineering metrics and models, Benjamin-Cummings Publishing Co., Inc., 1986.

[39] C. F. Kemerer and B. S. Porter, "Improving the Reliability of Function Point Measurement: An Empirical Study," *IEEE Transactions on Software Engineering*, vol. 18, no. 11, p. 1011–1024, 1992.

[40] J. Sprouls, "IFPUG Function Point Counting Practices Manual, Release 3.0," International Function Point Users Group, Westerville, Ohio, 1990.

[41] S. H. Kan, "Software Quality Metrics Overview," in *Metrics and Models in Software Quality Engineering*, *Second Edition*, 2002, p. 560.

[42] IFPUG(International Function Point Users Group), IFPUG Counting Practices Manual, Release 4.1, Westerville, Ohio, 1999.

[43] C. Jones, "Manual Software-Estimating Methods," in *Estimating Software Costs: Bringing Realism to Estimating*, Second Edition ed., The McGraw-Hill Companies, 2007, p. 644.

[44] IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries., New York, NY, 1990.

[45] D. Dörner, , The Logic of Failure: Recognizing and Avoiding Error in Complex Situations, Rowohlt Verlag GMBH, 1989.

[46] T. McCabe, "A Software Complexity Measure," *IEEE Transactions on Software Engineering*, vol. 2, pp. 308-320, 1976.

[47] M. N. Clark, B. Salesky, C. Urmson and D. Brenneman, "Measuring Software Complexity to Target Risky Modules in Autonomous Vehicle Systems," *AUVSI North America Conference*, pp. 1-11, 2008.

[48] K. E. Emam, The ROI from Software Quality An Executive Briefing, K Sharp Technology Inc., 2003.

[49] C. Jones, Software Assessments, Benchmarks, and Best Practices, Addison-Wesley, 2000.

[50] B. Littlewood and L. Strigini, "The risks of software," *Scientific American*, vol. 272, no. 3, pp. 180-185, 1995.

[51] M. Ann Neufelder, "The Naked Truth About Software Engineering," www.softrel.com, 2006.

[52] J. Burnim and K. Sen, "Heuristics for Scalable Dynamic Test Generation," *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pp. 443-446, 2008.

[53] S. Krishnamoorthy, "Tackling the path explosion problem in symbolic execution-driven test generation for programs," *Test Symposium (ATS)*, pp. 59-64, 2010.

[54] C. Cadar, D. Dunbar and D. Engler, "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs," *OSDI*, pp. 209-224, 2008.

[55] "The LLVM compiler infrastructure," [Online]. Available: http://www.llvm.org/.

[56] G. Li, I. Ghosh and S. Rajan, "KLOVER: A symbolic execution and automatic test generation tool for C++ programs," *Computer Aided Verification*, 2011.

[57] [Online]. Available: http://gcc.gnu.org/onlinedocs/gcc/Gcov.html .

[58] "uClibc++: An embedded C++ library," [Online]. Available: http://cxx.uclibc.org.

[59] L. Ciortea, C. Zamfir, S. Bucur, V. Chipounov and G. Candea, "Cloud9," *ACM SIGOPS*

*Operating Systems Review*, vol. 43, no. 4, p. 5, 2010.

[60] "VCC," [Online]. Available: http://vcc.codeplex.com/.

[61] "VCC," [Online]. Available: http://research.microsoft.com/en-us/projects/vcc/).

[62] R. DeLine and K. R. M. Leino, "BoogiePL: A typed procedural language for checking object-oriented programs," Microsoft Research,. Technical Report MSR-TR-2005-70, March 2005.

[63] L. d. Moura and N. Bjørner, "Z3: An efficient SMT solver," *Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008.

[64] M. Dahlweid, M. Moskal, T. Santen, S. Tobies and W. Schulte, "VCC: Contract-based modular verification of concurrent C," in *2009 31st International Conference on Software Engineering - Companion Volume*, 2009.

[65] "GYP," [Online]. Available: https://code.google.com/p/gyp/wiki/GypLanguageSpecification.

[66] P. Collingbourne, C. Cadar and P. H. Kelly, "Symbolic crosschecking of floating-point and SIMD code," in *Proceedings of the sixth conference on Computer systems - EuroSys '11*, New York, New York, USA, 2011.

[67] "metaSMT framework," [Online]. Available: http://www.informatik.uni-bremen.de/agra/eng/metasmt.php.

[68] H. Palikareva, C. Cadar, P. Hosek and K. Sen, "Multi-solver Support in Symbolic Execution".