

HOCHSCHULE RAVENSBURG-WEINGARTEN



CASE STUDY OF THE SOFTWARE DEVELOPMENT FOR SAFETY CRITICAL SYSTEMS

Report on Research Project

Remya Ramachandran

10/04/2010

CASE STUDY OF THE SOFTWARE DEVELOPMENT FOR SAFETY CRITICAL SYSTEMS

Hochschule	Ravensburg-Weingarten  University of Applied Sciences Ravensburg-Weingarten Hochschule Ravensburg-Weingarten
Study Program	Mechatronik 
Field of Research	Embedded Computing
Name	Remya Ramachandran
Matr.Nr.	20555
E-Mail	remyarmachandra@gmail.com
Supervisor	Mr. Philipp Ertle
Reviewer	Prof. Dr.-Ing. Holger Voos

CONTENTS:

1. INTRODUCTION -----4

2. OBJECTIVE-----5

3. SYSTEM DESCRIPTION-----6

4. FUNCTIONAL REQUIREMENTS-----7

 4.1. INPUT PARAMETERS-----7

 4.2. VIOLATION-----7

 4.3. ALERT LEVELS-----7

 4.4. REACTIONS-----7

5. FLOWCHART-----9

6. GUIDELINES TO BE ADAPTED FOR CODING-----11

7. FORMAL VERIFICATION OF CODE-----12

 7.1. COMMERCIAL TOOLS-----12

 7.2. OPEN SOURCE TOOLS-----13

 7.3. SPIN-----14

8. REFERENCES-----17

1. INTRODUCTION:

This project intends to study the software development process for the “Safety Board” installed in the mobile robot “Pioneer-3”. The Pioneer is a mobile robot from the company "ActivMedia Robotics". The Pioneer Family includes the mobile robot Pioneer 1, Pioneer AT, Pioneer 2-DX, DXF,-CE,-AT, Pioneer 2 --Dx8/Dx8 Plus and -At8/At8 Plus.

The Pioneer 3 have four pneumatic wheels and its maximum speed is 0.7 m / s. The drive has differential drive i.e. the left and right wheels are independently controlled. This makes it possible for the robot to rotate on the spot. The ultrasonic distance sensors of the robot enable it to estimate the surroundings. The robot has a solid aluminum casing and the basic equipments include DC motors, motor control and Drive electronics and a battery. It can be further equipped with camera or a laser scanner.

To test the robot can, the Demo -Program "Aria" included. This can be done from the PC using the arrow keys to move it in the intended direction.

The goal of the ZAFH robot is to learn small service tasks. For example: to bring coffee, to make tablecloths, articles and so on.



Fig: The ZAFH Mobile Robot “PIONEER-3”

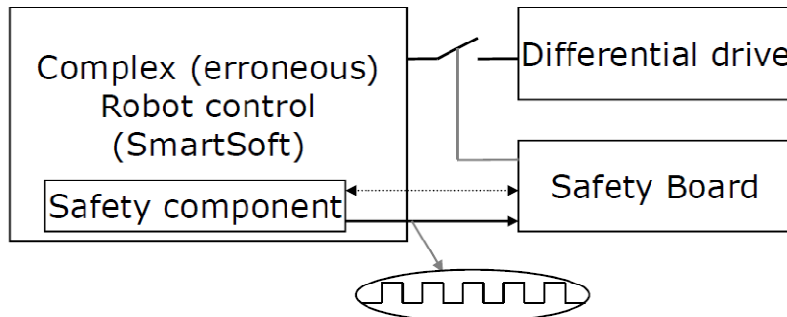
2. OBJECTIVE:

Robots are mechanical devices with potential for accomplishing much more than just the desired tasks. These machines can perform physical feats far in excess of human laborers. Unfortunately for humans in robot environments, the robot cannot think or 'realize' that damaging property and/or threatening human safety is not desirable. A moving robot arm has no concept of damage, and is not likely to stop before crashing through a retaining wall when its controlling program directs it to 'swing to the left through a full 270 degree arc.' It cannot possibly 'know' that a worker's hand protrudes between its lifting hook and the object to be lifted and will be crushed when the lifting action is performed. Such situations must be anticipated and software developed to accommodate these and similar hazards. The goal here is hence to conduct a case study into the software development process for a safety critical component like a differential drive for a service robot.

This study is specifically done on the Safety Board of the ZAFH Service robot. The safety board would enable the integration of autonomous robots in everyday life and it is fundamental to provide evidence that certain safety properties and constraints are guaranteed at any case. The principle aim of the software would be to ensure the redundancy and increased safety of the drive.

3. SYSTEM DESCRIPTION:

The safety Board provides a real time control for the safety factors by controlling the wheel drive of the robot. The following is a bird's eye view of the system.



The Safety Board installed currently makes only a “heartbeat query”. This is done parallel by a microcontroller and also via a redundant analogue circuit. Only when both the microcontroller and the redundant analogue circuit provide the correct output is the relay to the differential drive switched on. This Switching Relay is connected between the control board and motor. As long as the relay is not activated, no motor current flows. Consequently, the circuit is interrupted and the robot stops.

An emergency stop button for manual stop is also provided on the board. The motor PWM signal is taken before the "Motor-Power Distribution Board in order to realize a safe speed limitation.

4. FUNCTIONAL REQUIREMENT:

The Input signal to the Safety Board is a square wave signal provided from the Robot Control. Further on, it is described as the ‘heartbeat signal’. The software has to compare this heartbeat signal with the signal as expected when it is assumed that there is no error. Any variation in the parameters of the incoming signal from those of the standard signal is interpreted as ‘violations’. On detecting a violation, some corrective or precautionary actions (Reactions) are carried out and this information is also sent to the Robot Control.

The following terms are defined so as to make the functional requirements more clear, precise and quantitative.

4.1 INPUT PARAMETERS:

The input parameters are the variable values of the heartbeat signals that have to be constantly monitored for the detection of error/safety violation. These parameters are maximum level of the voltage (V_{min}), minimum level of voltage (V_{max}) and the frequency.

4.2 VIOLATIONS :

Violations are the variations in the Input signal from the standard signal. The violations are prioritized into levels 1, 2 and 3. Violations of severity level 1 should be acted upon with certain precautionary actions and if the violation persists for a defined time span ‘ t_1 ’ sec it is level is raised to level 2. On the occurrence of level 2 violations, certain reactions are taken and also it is checked if the violation is present after a time span ‘ t_2 ’ sec. If present, the level is raised to level 3. Level 3 violations should be immediately acted upon.

4.3 ALERT LEVELS:

Alert Levels contain the information about the state of the robot. They can be two levels Level 1 and Level 2. Level 1 means that the robot is in a “Warning” State while Level 2 means that the robot is in “Repair” state. A violation of Level 2 creates an alert of level2 and a violation for level 1 produces an alert of level1.

4.4 REACTIONS:

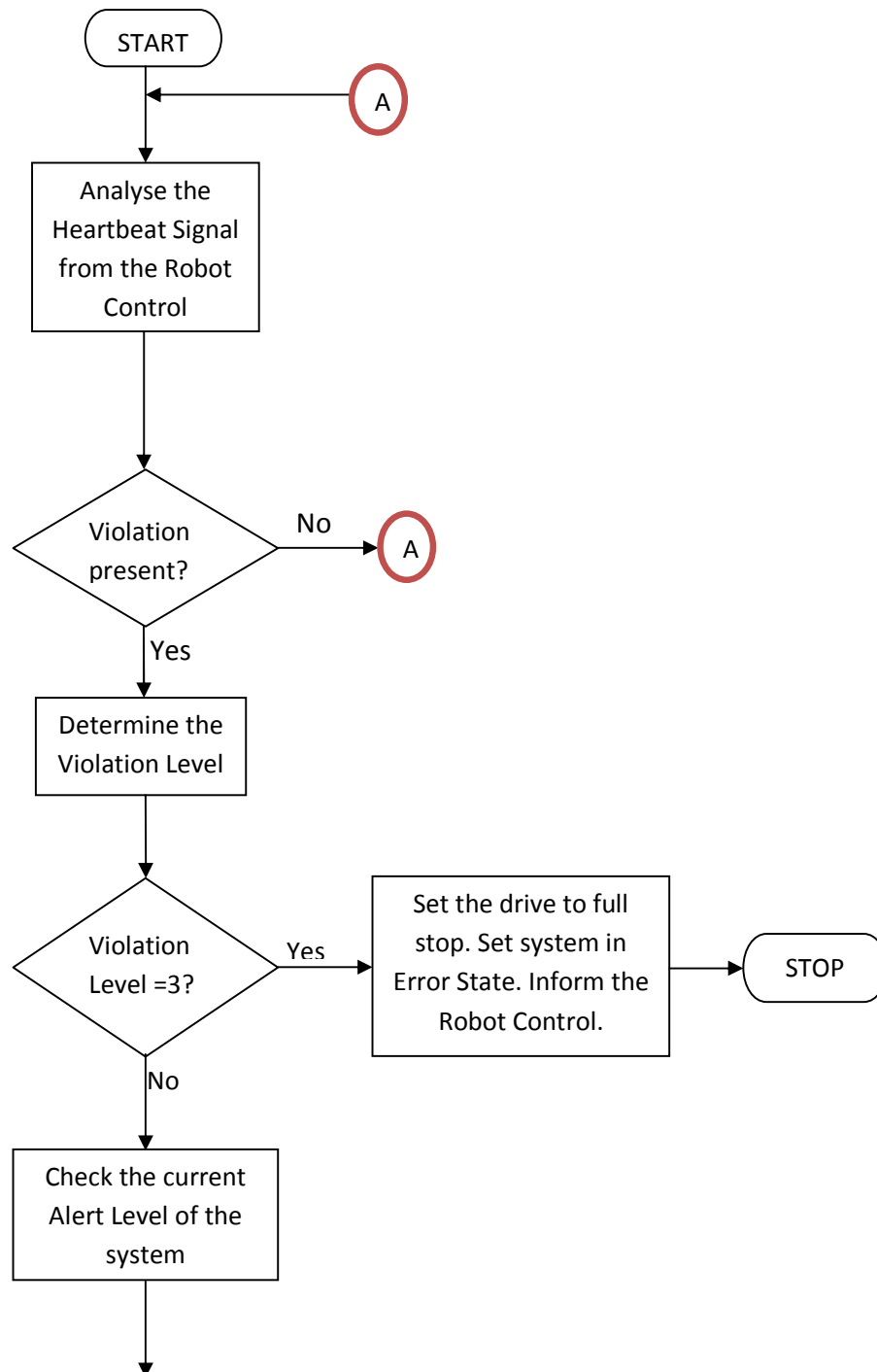
Reaction is the consequence of a violation, as shown in the Table 1.0

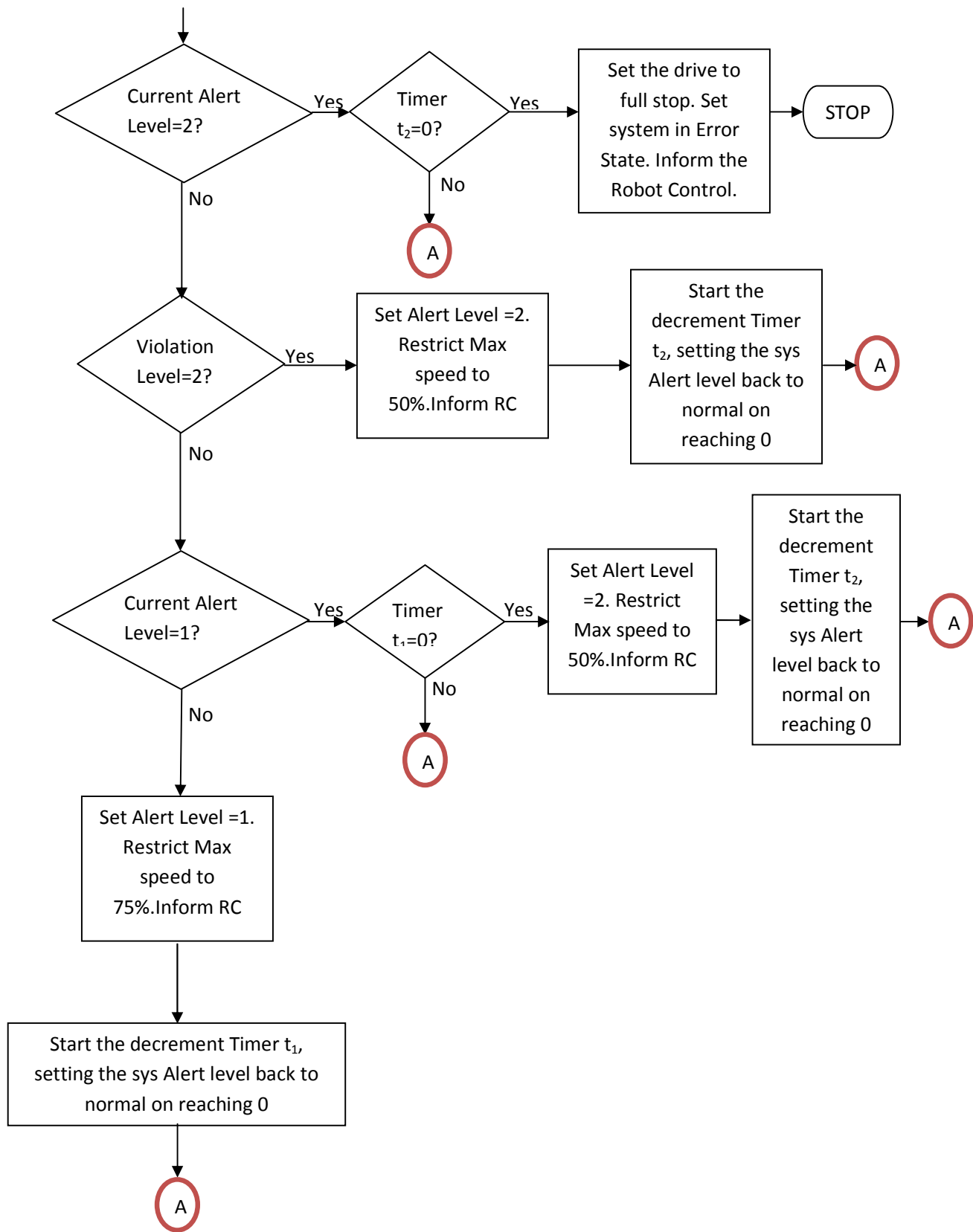
Violation	Alert Level	Reaction
Level 1	Level 1 <i>“Warning”</i>	Limit the max speed to 75% , Set Alert Level to 1, Inform the Robot Control
Level 2	Level 2 <i>“Repair”</i>	Limit the max speed to 50% , Set Alert Level to 2, Inform the Robot Control
Level 3	<i>“Error”</i>	Drive set to full stop, Inform the Robot Control

Table 1.0: Reactions expected for different Violations.

However, in the Version 1, the speed limiting functionality and the information passing are not realized. In case of a violation, the drive is set to a full stop and the alert level is set. In the Version 1.1, the information passing function is also realized.

5. FLOWCHART:





6. GUIDELINES TO BE ADAPTED FOR CODING:

The main aspect of many of the following guidelines is that it should allow for comprehensive tool-based compliance checks. Tool-based checks are important because manually reviewing the hundreds of thousands of lines of code that are written for larger applications is often infeasible.

The choice of language for safety critical code is in itself a key consideration. Here C is the preferred language because with its long history, it has extensive tool support, including strong source code analyzers, logic model extractors, metrics tools, debuggers, test-support tools, and a choice of mature, stable compilers. The following guidelines primarily target C and attempt to optimize the ability to more thoroughly check the reliability of critical applications written in C.

- (a) Restrict all code to very simple control flow constructs—do not use *goto* statements, *setjmp* or *longjmp* constructs or direct or indirect recursion. Simpler control flow translates into stronger capabilities for analysis and often results in improved code clarity.
- (b) All loops must be given a fixed upper bound. It must be trivially possible for a checking tool to prove statically that the loop cannot exceed a preset upper bound on the number of iterations. This can prevent the runaway code.
- (c) Dynamic Memory allocation is not to be used after initialization. Memory allocators such as *Malloc*, and Garbage collectors have unpredictable behavior that can significantly impact performance.
- (d) No function should be longer than what can be printed on a single sheet of paper in a standard format with one line per statement and one line per declaration. This implies that each function should be a logical unit that is understandable and verifiable as a unit.
- (e) Declare all data objects at the smallest possible level of scope. This is to enable the basic principle of data hiding.
- (f) The use of preprocessor must be limited to the inclusion of header files and simple macro definitions.
- (g) Assertions must be restricted to two per function. Assertion checks must be used to check for anomalous conditions that should never happen in real life executions. They should be side-effect free and should be defined as Boolean tests.
- (h) The use of pointers must be restricted. Specifically, not more than one level of dereferencing should be used.
- (i) Each calling function must check the return value of non void functions, and each called function must check the validity of all parameters provided by the caller.
- (j) All code must be compiled, from the first day of development and all code should compile without warnings. All code must also be checked daily with at least one strong static source code analyzers.

7. FORMAL VERIFICATION OF CODE:

In traditional methods, errors in hardware and software are discovered empirically, by testing them in maximum possible expected situations. However, this number of possible situations is normally so large that it is possible to only exercise a tiny proportion of them. Formal verification is an alternative that involves trying to prove mathematically that software code will function as intended.

Software reliability is a very crucial requirement for safety- and security-critical systems. To achieve this goal, code verification is one of the available options. In recent years, deductive code verification has improved to a degree that makes it feasible for real-world programs. Formal code verification methods are logic-based approaches to specify and validate the software (and hardware) systems, using a language with precise semantics, and usually including a technique for a mathematical verification of those properties. These formal methods have been proved to be useful in various stages of the software development process to improve the quality and reliability of software. Unlike the normal testing, formal software verification covers all possible inputs and every execution path of a program and, thus, is able to expose any error in the program w.r.t. a given specification.

In one approach, to check whether a program to be verified performs according to its specification, a logical formula is automatically generated from the source of the program and the specification. This formula, called *verification condition*, is rendered in predicate logic and has the property that, if it is valid, then the program is correct w.r.t. its specification. Finding a mathematical proof for the validity of this formula, which would serve as a witness for the correctness of the program, is then a task to be solved by a theorem proving system.

In another approach, the system can verify some of the properties with the help of an exhaustive search of all possible states it could enter during its execution. Known popularly as Model Checking, it checks the hardware and software models whose specification is given by a temporal logic formula.

7.1 COMMERCIAL TOOLS:

The major commercial players in the area of Formal Verification tools are:

(a) CodeSonar (by Grammatech)

It is an effective tool for spotting code defects and suspicious code fragments. The tool has been extended with a rule checker for Power of Ten coding rules for safety critical code, which makes it attractive for high integrity applications. This one is especially good at inter-procedural analysis. It can be slow on large code bases, but is quite thorough and accurate.

(b) Coverity

A popular tool based on Dawson Engler's methodology for source code analysis of large code bases. An extended version of the tool (Coverity Extend) supports user-defined properties. The tool is fast and returns few false positives, but it can be expensive.

(c) KlocWork

It provides support for static error detection, with added project management and project visualization capabilities. Fast, almost as thorough as Coverity, and not quite as expensive. The tool is especially good at finding array bound violations. A capability for user-defined checks is pending.

(d) PolySpace

Polyspace, from TheMathWorks.Inc, claims it can intercept 100% of the runtime errors in C programs. Main customers of this tool are in the airline industry and the European space program. No test cases, instrumentation or execution is required. Can be thorough, but also very slow, and does not scale beyond a few thousand lines of code. Does not support full ANSI-C language (e.g., it places restrictions on the use of *gotos*).

(e) PRefix and PRefast (by Microsoft)

PRefix was developed by Jon Pincus; MicroSoft acquired the tool when it bought Pincus' company. PRefast is a lighter weight tool, developed within Microsoft as a faster alternative to PRefix (though it is said not to be directly based on PRefix). Both tools are reported to be effective in intercepting defects early, and come with filtering methods for the output to reduce false positives. PRefast allows for new defect patterns to be defined via plugins. Less than 10% of the code of PRefix is said to be concerned with analysis per se, most applies to the filtering and presentation of output, to reduce the number of false positives.

7.2 OPEN SOURCE TOOLS:

Some very good Open Source tools are also available. The very popular ones are:

(a) Spin

Spin (starting with version 4) provides direct support for the use of embedded C code as part of model specifications. Spin supports a high level language to specify systems descriptions, called PROMELA (a PROcess MEta LAnguage). To verify a design, a formal model is built using PROMELA, Spin's input language. Also, Spin is actively maintained and continuously improved and updated. It is been claimed by the Spin developers that selected algorithms for a number of space missions include Deep Space 1, Cassini, the Mars Exploration Rovers, Deep Impact, etc. were verified with the Spin model checker.

(b) Splint

Short for Secure Programming Lint, Splint is a programming tool for statically checking C programs for security vulnerabilities and coding mistakes. Formerly called LC Lint, it is a modern version of the UNIX lint tool. Splint has the ability to interpret special annotations to the source

code, which gives it stronger checking than is possible just by looking at the source alone. Splint is free software released under the terms of the GNU General Public License.

(c) Penjili (by EADS Innovation Works)

The tool mainly targets the memory manipulation bugs (array, pointer and stringout-of-bounds) and arithmetic bugs (integer overflows and division by zero). It can compile most ANSI C (no backward *gotos*) namely conditional expression, bitfields, and variable number of arguments. Also compiles some GNU C. It claims to compile 40 million lines of macro-expanded C in 9 minutes.

(d) C Code Analyzer – CCA

No code annotations or tweaking is required - it's fully automatic. CCA tries to spot only the errors that can actually cause problems. CCA is licensed under a BSD license. The current features are: fully automatic user input tracer, memory leak detection, multiple/dangling free detection, array out of bound accesses and potential buffer overflow detection.

(e) Clang Static Analyzer

The Clang Static Analyzer is source code analysis tool that find bugs in C and Objective-C programs. Currently it can be run either as a standalone tool or within Xcode. The standalone tool is invoked from the command-line, and is intended to be run in tandem with a build of a code base. The analyzer is 100% open source and are part of the Clang project.

A detailed study about the Formal Verification tool “Spin” has been carried out as it is not only an open source tool but also has comparatively better documentation support and a large number of internet forums discussing the usage of the tool.

7.3 SPIN:

Some of the features that set Spin apart from related verification systems are:

Spin targets efficient software verification, not hardware verification. The tool supports a high level language to specify systems descriptions, called PROMELA (a PROcess MEta LAnguage). Spin has been used to trace logical design errors in distributed systems design, such as operating systems, data communications protocols, switching systems, concurrent algorithms, railway signaling protocols, etc. The tool checks the logical consistency of a specification. It reports on deadlocks, unspecified receptions, flags incompleteness, race conditions, and unwarranted assumptions about the relative speeds of processes.

Spin (starting with version 4) provides direct support for the use of embedded C code as part of model specifications. This makes it possible to directly verify implementation level software specifications, using Spin as a driver and as a logic engine to verify high level temporal properties.

Spin (starting with version 5) provides direct support for the use of multi-core computers for model checking runs -- supporting both safety and liveness verifications.

Spin works on-the-fly, which means that it avoids the need to preconstruct a global state graph, or Kripke structure, as a prerequisite for the verification of system properties.

Spin can be used as a full LTL model checking system, supporting all correctness requirements expressible in linear time temporal logic, but it can also be used as an efficient on-the-fly verifier for more basic safety and liveness properties. Many of the latter properties can be expressed, and verified, without the use of LTL.

Correctness properties can be specified as system or process invariants (using assertions), as linear temporal logic requirements (LTL), as formal Büchi Automata, or more broadly as general omega-regular properties in the syntax of never claims.

The tool supports dynamically growing and shrinking numbers of processes, using a rubber state vector technique.

The tool supports both rendezvous and buffered message passing, and communication through shared memory. Mixed systems, using both synchronous and asynchronous communications, are also supported. Message channel identifiers for both rendezvous and buffered channels, can be passed from one process to another in messages.

The tool supports random, interactive and guided simulation, and both exhaustive and partial proof techniques, based on either depth-first or breadth-first search. The tool is specifically designed to scale well, and to handle even very large problem sizes efficiently.

To optimize the verification runs, the tool exploits efficient partial order reduction techniques, and (optionally) BDD-like storage techniques.

To verify a design, a formal model is built using PROMELA, Spin's input language. PROMELA is a non-deterministic language, loosely based on Dijkstra's guarded command language notation and borrowing the notation for I/O operations from Hoare's CSP language.

Spin can be used in four main modes:

- As a simulator, allowing for rapid prototyping with a random, guided, or interactive simulations
- As an exhaustive verifier, capable of rigorously proving the validity of user specified correctness requirements (using partial order reduction theory to optimize the search)
- As a proof approximation system that can validate even very large system models with maximal coverage of the state space.

- As a driver for swarm verification (a new form of swarm computing), which can make optimal use of large numbers of available compute cores to leverage parallelism and search diversification techniques, which increases the chance of locating defects in very large verification models.

All Spin software is written in ANSI standard C, and is portable across all versions of Unix, Linux, cygwin, Plan9, Inferno, Solaris, Mac, and Windows.

8. REFERENCES:

- Better Avionics Software Reliability by Code Verification; Christoph Baumann, Bernhard Beckert ,Holger Blasum and Thorsten Bormer Available at <http://www.uni-koblenz.de/~beckert/pub/embeddedworld2009.pdf>
- Principles of Program Analysis ,F. Nielson, H. R. Nielson and C. Hankin, Writing Solid Code, Steve Maguire, Microsoft, 1993.
- Code Complete, Steve McConnell, Microsoft, 1993.
- The Practice of Programming, Kernighan & Pike, Addison-Wesley, 1999.
- C Programming Language (2nd Edition), Kernighan & Ritchie, Prentice Hall, 1988.
- Beautiful Code: Leading Programmers Explain How They Think (Theory in Practice (O'Reilly))
- The Power of 10:Rules for Developing Safety-Critical Code ,Gerard J. Holzmann; NASA/JPL Laboratory for Reliable Software
- Internet Websites of the Home pages of the Open source and Commercial Tools.
POLYSPACE <http://www.mathworks.de/products/polyspace/>
VARVEL <http://www.nec.co.jp/techrep/en/journal/g07/n02/070209.html>
SPIN <http://spinroot.com/spin/whatispin.html>
SPLINT <http://splint.org/>
CODESONAR <http://www.grammatech.com/products/codesonar/>
KLOCWORK <http://www.klocwork.com/products/insight/>
COVERITY <http://www.coverity.com/>
PENJILI <http://www.penjili.org/penjili-tool.html>
CLANG <http://clang.llvm.org/>