Safety by Design for the Mariokart System

Simon Richards
scr52@uclive.ac.nz

Coauthors: Wim Looman, Zachary Taylor Henry Jenkins and Dr Andrew Bainbridge-Smith
wgl18@uclive.ac.nz zjt14@uclive.ac.nz hvj10@uclive.ac.nz
andrew.bainbridge-smith@canterbury.ac.nz

Department of Electrical and Computer Engineering
University of Canterbury
Christchurch, New Zealand

***Abstract*—An electric go-kart is outfitted with a drive-by-wire system in the first phase of development of an autonomous vehicle. The safety aspects of the design are evaluated and analysed using typical project management techniques. A model of the distributed software required is developed and proven and the place of modelling in software engineering is discussed..**

I. Introduction

A. Project Description

The authors' reports are based on a group project carried out in the final year of their Bachelor of Engineering degree at Canterbury University. The original goal of this project was to take one of the electric go-karts used by the department for power electronics projects and convert it into an autonomous vehicle capable of a simple navigation task. Due to time restrictions the aim was reduced to implementing the mechanical and electronic systems needed to control the kart's motion over a USB link.

The authors refer to the project using the moniker Mariokart and this name may be used throughout this report in reference to the team, the vehicle or the project itself.

B. System Design

The overall system design consisted of a laptop, electronics, a CAN bus, actuators and an interface to the existing motor driver hardware.

- Electronics: Five PCBs were located around the kart, each in close proximity to the sub-system it was responsible for. The PCBs were identical in layout, specialisation was achieved by removing components from boards where they were not needed. For example it was intended for only one board to communicate directly with a laptop and so the other boards did not have USB headers. The decision to distribute control was made primarily to avoid problems caused by driving motors and actuators over a long distance in a high noise environment.
- Actuators: The brake pads were driven by a linear actuator mounted in place of the brake pedal. The actuator was mounted to the floor of the go-kart and its driver board was situated nearby. The steering column was similarly controlled by a DC motor mounted in place of the steering wheel and fixed to the frame. The DC motor included an encoder so that the relative angle of the steering column could be determined. Limit switches at each extreme of rotation ensures the motor does not turn too far and allows for the absolute angle to be measured.
- Controller laptop: To simplify use for future projects, the entire system is controllable via commands sent over a USB cable. The communications board is responsible for interfacing with a laptop and distributing the user's commands
- Motor interface: Power electronics and a motor driver had already been implemented as part of

a student project so the only requirement from the motor driver board was to interface with this over the available SPI link.
- The CAN bus: CAN is a multi-master serial bus with guaranteed sending and message priority. It was designed by Robert Bosch GmBH for networking micro-controllers in automobiles and has also some popularity in industrial automation. The CAN specification is low-level only, it does not specify a data protocol or a method address allocation. There are numerous similar software stacks built on top of CAN for various purposes such as the Society of Automotive Engineers' J1939 [1] however due to the simplistic nature of our network we implemented our own protocol.

For a more detailed description and justifications see *Embedded Hardware Design For Autonomous Electric Vehicle* [2].

C. Risk

As with any vehicle, autonomous or otherwise, there is always a risk to the user, bystanders and the go-kart itself if a part of the system malfunctions in any way. Replacing a human driver with electromechanical control systems removes the non-deterministic element of human error from the system but adds a new set of risks and potential design mistakes.

In this paper, the sources of these risks will be listed analysed and techniques for mitigation and negation will be investigated and evaluated.

II. Safety - A Method

Risk, according to ISO 31000 [3], is the effect of uncertainty on objectives. Therefore, in order to minimise risk we need to minimise uncertainty.

In practical terms this means analysing the system from the top down, creating a hierarchy of risks to ensure that all issues are considered. Taking this approach means that even if the engineer does not identify every possible cause of failure (which is very likely), mitigation of the failure modes not identified may still be achieved via global mitigations which are applied to an entire branch of the taxonomy. When applying a divide and conquer approach as shown in figure1 we only need to ensure that each failure set is equal to the union of those below it.

At the lowest level of figure1 the categories are colour coded into the following groups:

G Failures that are not a source of risk as they may be eliminated during testing.

R Failures that are not preventable but may warrant mitigation.

B A failure which could happen unpredictably and could be more severe than loss of power. Preventing this from occurring is the main focus of this paper.

A. Preventing Mistakes (Green)

For a detailed look into software and systems engineering practices see Looman's *Software Engineering in the Mariokart System* [4]. If standard design practices such as unit testing are followed then it is possible to eliminate risk from these sources.

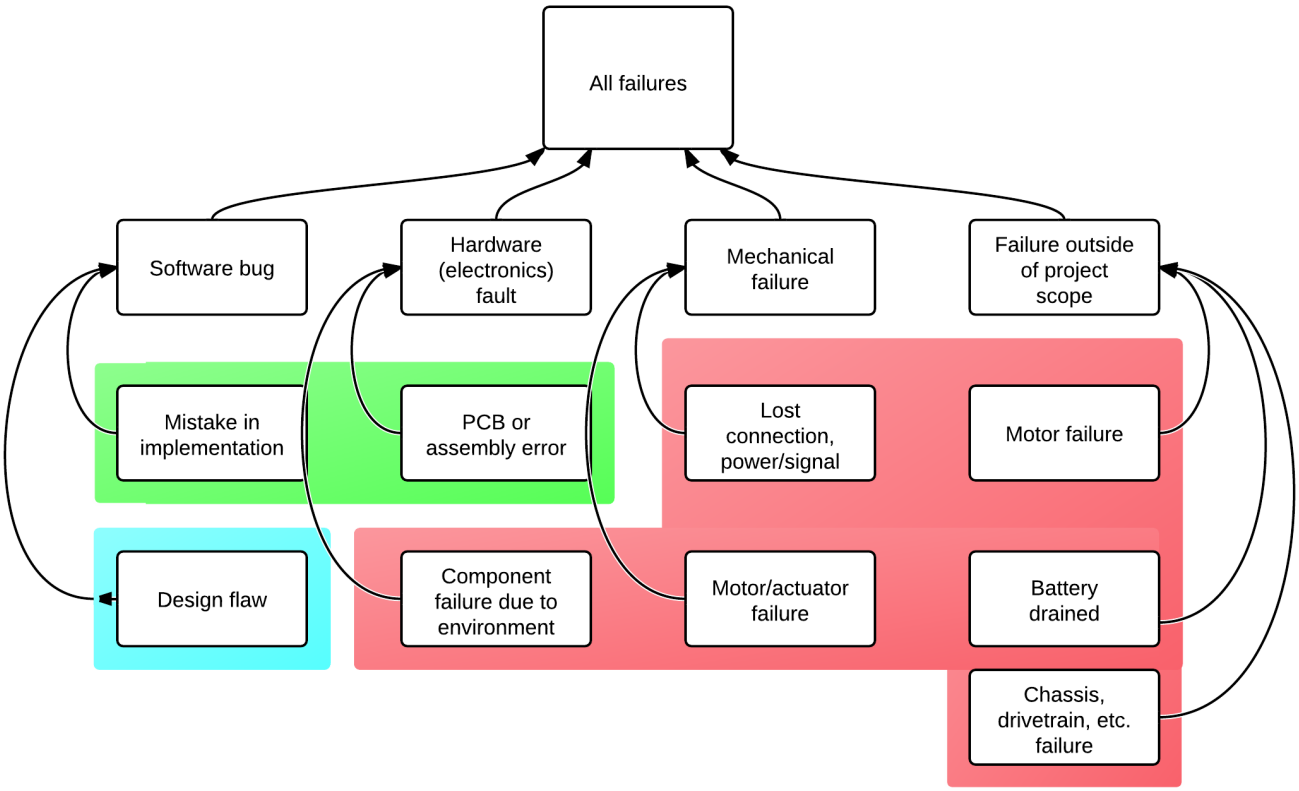
B. Risk Analysis (Red)

Traditional project management tools for risk management involve tools like Risk Matrices and qualitatively assessing the potential consequences and probabilities of risks. Such tools are popular mainly for their simplicity and have little backing in research, taking criticism from researchers such as Tony Cox [5] for reasons such as their qualitative nature and reliance on experienced judgement. Cox even goes as far to suggest that in some cases using Risk Matrices leads to worse-than-random decisions.

For this reason and for simplicity's sake it was decided to lessen the consequences of potential failure modes with a set of general mitigating techniques. Hence we have the following set of guidelines and design decisions:

- Despite the goal of autonomous function, a passenger should always be in the vehicle during operation so that the vehicle may be shut down immediately by the provided kill switch.
- All subsystems fall into an error state immediately after communication is lost with any board. Any board losing then regaining power will also transition into an error state.
- In the error state each board will perform whatever is necessary to minimise the potential

Fig. 1. A hierarchy of failures.



harm in case the go-kart is currently mobile. The important detail here is that both the brake board and motor will attempt to slow the vehicle (the motor is capable of regenerative braking) providing redundancy of the most critical function.

C. Automated model checking to prevent design errors (Blue)

While it is a given that a structural engineer will calculate forces and stress before starting constructing and that a mechanical engineer may likewise simulate loads on his or her design using a CAD package. Software engineering has, due to the relative flexibility of the medium, been exempt from the expectation that a design must be proven before it is implemented. This is not necessarily an advantage, however, as modelling can provide benefits including but not limited to proof of system robustness.

SPIN is an automated model checker for 'the formal verification of distributed software system.' [6] The name is an acronym for Simple Promela INterpretor, where Promela is the language used to specify a model and its constraints. For more information on SPIN and Promela visit spinroot.com but for the purposes of this paper an in-depth knowledge is not required.

When modelling a software system, heavy-handed abstraction is useful and sometimes necessary. This is because accurately modelling the software to the level of calculations that have no effect on the system state only serves to increase the state-space and therefore the amount of computation required to check the model.

Conversely too much abstraction will limit the usefulness of a model to being little more than a way of sketching out the top level design of a distributed system.

Hence we wish to find an optimal level of detail which will lie somewhere between the overly naive and ultimately useless model which assumes too

much; and the result of several years of work and many hours of computation which simulates a system down to the instruction set level.

The level of detail which is right for a project is subject to the law of diminishing returns, where an increase in the severity of failure or the resources available to the project will justify a more in depth model to be developed.

1) The (other) Benefits of Modelling a Distributed System: The obvious aim of modelling is model checking, proving that a design is correct. That is not the only benefit that modelling brings to a project however.

Writing a model before code provides the author with a 'sketched' version of the code. Unlike a state diagram sketched on paper however the Promela sketch can be simulated visually using the iSpin frontend for spin. iSpin is capable of generating sequence diagram-like output showing message transmission and reception between processes as well as other, more complicated graphical output we won't discuss here.

D. Modelling Mariokart

The software for mariokart was modelled with a focus on the state machine seen in appendix B and the message passing between boards. Failure was explicitly allowed as seen in the first part of appendix A where the client board may non-deterministically (the :: operator) choose to either reset itself, discover an error or continue running as expected. Notice the probabilities of these three events occurring is not specified as SPIN will exhaustively explore all possibilities regardless of their respective probabilities.

The constraints placed on the model enforce a consistent state between the boards and ensure that the system either executes infinitely or all boards reach an error state and exit. Although the second option is not desirable in practice, showing that the boards fail gracefully regardless of the failure mode is the most important part of the model as it proves the correctness of an aspect of the system that would be difficult or maybe even impossible to test manually.

As previously explained, if the team had more resources or if the project came with significantly more danger (for example if the authors were modifying a space shuttle instead of a go-kart) then a more in depth model would be justified.

As it stands, however, the time restrictions imposed on the project led to the relatively simple model described above being developed quickly before being implemented in C.

III. Discussion

A responsible engineer evaluates his or her design before implementing it and it is the author's opinion that this belief is sometimes lacking in software development. Since a Promela model skips the details of implementation it is simple to write and understand and does not bury the top level design in details. If written before implementation or concurrently it simplifies writing the actual software as the design is already specified and proven.

Balancing the cost of developing the model with the benefits it can bring is an important consideration for any team wishing to travel the same path as the authors did with mariokart. For groups or individuals working on projects similar to the authors' it is recommended to at least explore modelling top level designs before implementing them. The worst case outcome is the cost of learning a new language (and Promela contains few difficult concepts) and the potential benefits include quicker development times and the chance to catch a potentially dangerous fault before it can manifest itself.

IV. Conclusion

The potential for failure in the mariokart system was analysed. Solutions, mitigations and justifications for ignoring potential problems were described along with the costs and benefits of modelling software.

References

[1] SAE, 2011. [Online]. Available: http://www.sae.org/standardsdev/groundvehicle/j1939.htm

[2] H. V. Jenkins, "Embedded hardware design for autonomous electric vehicle," 2011. [Online]. Available: https://raw.github.com/team-ramrod/mariokart/master/Documentation/ScientificReport/Henry/report.pdf

[3] ISO, 2009. [Online]. Available: http://www.iso.org/iso/catalogue_detail.htm?csnumber=43170

[4] W. G. Looman, "Software engineering practices in the mariokart system," 2011. [Online]. Available: https://raw.github.com/team-ramrod/mariokart/master/Documentation/ScientificReport/Wim/report.pdf

[5] A. Cox Jr *et al.*, "What's wrong with risk matrices?" *Risk Analysis*, vol. 28, no. 2, pp. 497–512, 2008.

[6] (2011) Spin - formal verification. [Online]. Available: http://www.spinroot.com/

V. Appendices

A. Promela examples

Client Board proctype:

```
/**
 * A greatly simplified version
 * of the client board model.
 */
proctype Client(chan input) {
Startup:
    input?req_transition;
    comms!ack_transition;

Calibration:
    // calibration handling
    // More message passing
Running:
    do
        :: //Normal routine
        :: goto Startup
        :: goto Error
    od;
Error:
    broadcast!error
}
```

Never claim: All boards transition to error state or remain in running state indefinitely:

```
/**
 * If one board goes into
 * error state, they all must.
 */
never {
    do
        :: error_count > 0 -> break
        :: true -> skip
    od;

accept:
    do
        :: error_count != num_boards
    od;
}
```

B. Single board state machine

Fig. 2. The state machine each board follows.