Code Craft

Gerard J. Holzmann

Which is harder: overcoming the many physical challenges of autonomously landing a spacecraft on the surface of a distant planet, building the many interacting pieces of hardware to execute such a mission, or writing the software that controls the whole process? To me, the first two tasks look impossibly hard. But, if we look at a series of failed attempts to accomplish this feat, it may well be that writing the software correctly is still the harder problem.

The recent loss of the Schiaparelli spacecraft from the European Space Agency, now thought to have been caused by a software defect, is a powerful reminder of this fact, though it is certainly not the only one, nor is it likely to be the last. Why is it so hard to write bug-free code? Over half a century ago we hopefully started calling our discipline 'software engineering,' but how much do we really have in common with more traditional engineering disciplines?

Quite a few people have tried to make a comparison between the design processes that are used in software engineering and in civil engineering.[1] The comparison is usually made in a somewhat accusatory way, questioning why software engineers make such scant use of tools that could help them model and analyze software systems before they are implemented. It would be unthinkable for a civil engineer to propose building a structure before its design was carefully vetted in analytical models, audited for compliance with all the relevant standards, and checked for respecting proper safety margins. Why is this still so different in software engineering?

> ***Building software is perhaps closer to the construction of a sand-castle at the beach, or playing a game of sticks, than it is to designing a bridge or a house.***

I will cross my fingers that not too many actual civil engineers will read this column, because they would probably disagree with the following statement: it may be easier to model and analyze the design of a physical structure than it is to model and analyze the reliability of a complex software system. I say this because a large bridge or a tall building is unlikely to come tumbling down if a single rusty nail hiding somewhere deep into its internal structure unexpectedly breaks. Paint peeling off at one end of a bridge is similarly unlikely to cause the roadway to collapse at the other end. Yet, this is the type of thing that can happen in large software systems. The often made analogy between the construction of a bridge or house and the construction of a software system is therefore flawed. Building software is perhaps closer to the construction of a sand-castle at the beach, or playing a game of sticks, than it is to designing a bridge or a house. Building sand-castles instead of concrete structures requires different types of skills and tools.

Robustness

Even though many of us feel that hardware systems can be more robust than software systems, there is an enduring trend to move functionality gradually away from hardware and into software, even in safety

critical systems. Software is clearly simpler to update or extend than hardware, and we all want to believe that with sufficient care software can be just as reliable. There are many reminders that justifying this may require more evidence than is often assumed. Building robust software really does require a different approach.

Therac-25

The Therac-25 radiation therapy machine is perhaps one of the best known examples of the danger of blindly moving critical safety protection mechanisms from hardware to software.[2] The Therac-25 was introduced in the early eighties as an improvement over two successful earlier machines, the Therac-20 and Therac-6. The earlier machines had originally been developed with minimal software support, relying on solid hardware interlocks for their safety features. The new machine was instead designed from the start to be software controlled.

Between June 1985 and January 1987 six cases of massive overdoses were accidentally caused by the new machine, resulting in serious injuries and in some cases in the death of patients. Although there were several contributing causes to these accidents, a few stand out. One reported cause was the existence of a race condition in the code that allowed the machine to apply radiation at a much higher level than the operator intended. The effect of the race condition depended on the speed with which proficient operators could enter settings into the machine, and the ease with which they could override inscrutable error codes.

It was later found that some of the critical design defects in the software had also been present in the earlier Therac-20 and Therac-6 machines, but the independent hardware interlocks that were used in those machines had prevented accidents. The replacement of the hardware interlocks with software versions, combined with misleadingly simple software overrides for warnings, had inadvertently introduced the new risks. The manufacturer of the machine initially denied that the overdoses could possibly have been caused by the machine, which they believed had been well-tested. It later transpired that the actual testing had been minimal, and that virtually no meaningful analysis had been performed on the safety of the reused software.

Generally available tools for detecting race conditions and data corruptions errors in software systems did not yet exist at the time that the Therac-25 software was developed, but that changed soon after. In the next example I'll show how tools that have been available for about a quarter century now can be used to thoroughly analyze both hardware and software subsystems for critical safety properties.

Hardware Interlocks

A former colleague, Peter van Eijk, once published an example a fairly straightforward model checking analysis of a hardware lockout circuit that was used in trains.[3] The relay circuit was used to ensure that trains with two driver cabins, one on each end of the train, could not be operated from both cabins at the same time.

A relay is basically an electro-magnet that can open or close switches when it is connected to a power source. By using a combination of relays in an electrical circuit one can design fairly complex systems

that are not all that easy to analyze with traditional means. Van Eijk first described how the working of a system of switches and relays can be described with simple Boolean formula.

For example, the circuit shown in Figure 1 contains three switches, that I've marked a, b, and c, and a single relay marked A. In Figure 1, switch c is closed, and switches a and b are open. The horizontal line at the top of the figure represents a power source, and the horizontal line at the bottom represents ground. Relay A will be powered when simultaneously switch b is closed and either of the two switches a or c is also closed. We can express that in a formula symbolically as: A = b and (a or c).

We now add one final wrinkle to the functioning of this circuit. We'll use the convention that every switch that is controlled by a relay carries the same letter as that relay, but in lower-case. So, in Figure 1, switch a is controlled by relay A: if A is powered then switch a will close. This makes sure that once relay A is powered it can remain so independent of the position of switch c.

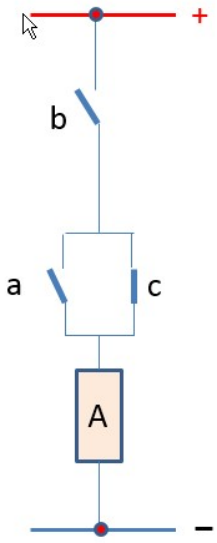

Figure 1 – Example of a simple relay corresponding to the formula A = b and (a or c).

Figure 2 shows an extended version of this simple circuit, as it was used to implement a hardware lockout between a front- and rear train cabin.

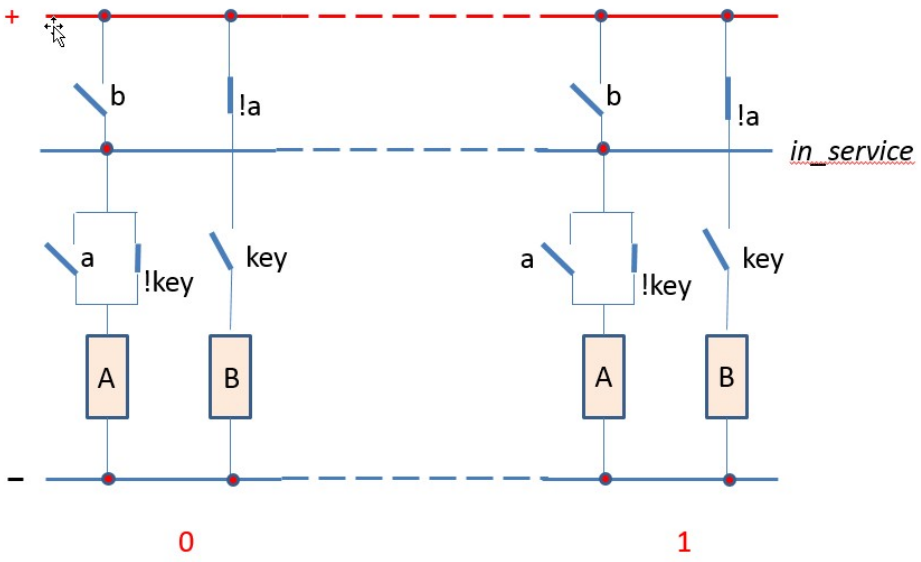

Figure 2 – Relay circuit used as a hardware lockout for trains. In trains that can be controlled from both the front (left) and rear (right) cabins only the first key inserted by a driver should succeed in powering the B relay on that side, which then supplies power to the "in_service" wire.

In Figure 2 we see two copies of a small extension of the circuit from Figure 1, both connected to the same power and ground. The circuit adds an additional horizontal wire, called in_service, that is powered when at least one of the switches a or b is closed. We use the notation !a to indicate that a switch controlled by relay A is closed instead of open in the unpowered state.

To avoid confusion we'll distinguish the various switches in the two copies of the circuit with an index. For example, we'll use a[0] to refer to switch a in the left-hand copy and a[1] to refer to the same switch in the right-hand copy.

There are four relays in the new circuit, two on each side. There are also four switches marked with either key or !key. These switches are controlled by an operator inserting a key in either the front or the rear cabin of the train. Figure 2 shows all switches in the unpowered state, with neither key inserted. When the B relay on either side is powered, finally, that side of the train will control the train, so it is important that both B relays cannot be powered simultaneously.

Driving the Train

Let's look at what happens when the driver in the front cabin inserts a key to take control of the train. The switch key[0] will close, and simultaneously switch !key[0] will open. That means that relay B receives power and cause switch b[0] to close. Relay A[0] stays unpowered, because both switches a[0] and !key[0] are now open. But, the in_service wire is now powered, which means that relay A[1] in the other cabin will be powered as well. This in turn closes switch a[1] and opens !a[1]. Because switch !a[1] is now open, inserting the driver's key in the rear cabin will have no effect. Similarly, because switch a[1] is now closed, relay A[1] will stay powered independent of whether the second key is present or not. So, all seems to work as intended with this circuit. But is it really reliable?

It is not terribly complicated to capture the definitions of the state of each switch in Boolean formula like we did for Figure 1. For relay A[0], for instance, we can write:

```
A[0] = (b[0] or in_service) and (a[0] or !key[0])
```

It is straightforward to build a small Spin model that can mimic the functionality of the entire circuit, including the possible actions of the train operator(s) inserting or removing their key. The model is shown in Figure 3.

```
bool a[2], b[2], key[2], in_service

active [2] proctype cabin()
{    bool s = _pid

     do
     :: a[s] = (b[s] || in_service) && (a[s] || !key[s])
     :: b[s] = key[s] && !a[s]
     :: in_service = b[s] || b[1-s]
     :: key[s] = !key[s]
     :: assert(!b[s] || !b[1-s])
     od
}
```

Figure 3 – Small Spin verification model to check the correctness of the circuit in Figure 2.

The Spin model defines two processes called cabin. The two copies are distinguished by a unique process identification number, or _pid, that is either 0 or 1 in this case. We use that _pid (assigned to variable s) to set the correct value for each of the a and b switches, and the state of the in_service wire.

The 4th line in the do-loop in this small model indicates that the driver in either cabin can arbitrarily insert or remove their key. The 5th line, finally, contains an assertion that states that the b relays cannot both be powered (and thus the b switches closed) at the same time.

To analyze the correct functioning of the circuit, all we have to do is now run the model checker and ask it if there is any possible scenario that could lead to the failure of the assertion. The following command runs the analysis, using a breadth-first search method to get the shortest possible trace to an assertion violation, in case one is possible.

```
$ spin –run –bfs train.pml
assertion violated ( !(b[s])|| !(b[(1-s)])) (at depth 4)
wrote train.pml.trail
```

The search completes virtually instantly, finding an assertion violation after just four steps. We can replay that scenario as follows, printing every process step that is executed, and ending with the resulting settings of all switches.

```
$ spin -p -replay train.pml
using statement merging
  1:    proc  1 (cabin:1) train.pml:8 [key[s] = !(key[s])]
```

```
  2:    proc  1 (cabin:1) train.pml:7 [b[s] = (key[s]&&!(a[s]))]
  3:    proc  0 (cabin:1) train.pml:8 [key[s] = !(key[s])]
  4:    proc  0 (cabin:1) train.pml:7 [b[s] = (key[s]&&!(a[s]))]
  5:    proc  1 (cabin:1) train.pml:10[assert()]
spin: train.pml:10, Error: assertion violated
spin: text of failed assertion: assert((!(b[s])||!(b[(1-s)])))
spin: trail ends after 5 steps
#processes: 2
                a[0] = 0
                a[1] = 0
                b[0] = 1
                b[1] = 1
                key[0] = 1
                key[1] = 1
                in_service = 0
  5:    proc  1 (cabin:1) train.pml:5 (state 6)
  5:    proc  0 (cabin:1) train.pml:5 (state 6)
2 processes created
```

In this scenario first driver 1 inserts a key, thereby closing switch key[1], and opening !key[1]. Next, relay B[1] is powered, which closes switch b[1]. That in turn will apply power to the in_service wire, but that step isn't recorded in the scenario yet. Before the circuit can stabilize further, driver 0 now inserts the second key, closing switch key[0] and opening !key[0]. This powers relay B[0] and closes switch b[0]. If switch !key[0] opens before b[0] closes, relay A[0] remains unpowered, and switch !a[0] remains closed. Similarly on the right hand side of the circuit, relay A[1] remains unpowered and switch !a[1] remains closed, leaving the circuit in a stable state with both drivers in control of the train.

The scenario reveals a race condition in the circuit that would be hard to spot without the use of tools. It is not too hard to fix the circuit to prevent these types of race conditions from having ill-effects, once we know that it's not quite right yet. It is increasingly the case that for both hardware and software design, not using the right analysis tools is just not cool.

References

[1] Leslie Lamport, Who builds a house without drawing blueprints? Communications of the ACM, Vol 58, No. 4, pp. 38-41. (2015)

[2] Nancy Leveson and Clark Turner (July 1993). *"An Investigation of the Therac-25 Accidents"* (PDF). IEEE Computer. **26** (7): 18–41.

[3] Peter van Eijk, Verifying relay circuits using state machines. Proc. 3rd Spin Workshop, Twente University, The Netherlands, April 1997.

[4] Source and binary versions of the Spin model checker are freely available from: http://spinroot.com/