Driving Efficiency & Resilience to Human Error: Safecap Automated Verification of Signalling Data

Dominic Taylor, CEng MBA, SYSTRA Scott Lister
Alexei Iliasov, PhD, Newcastle University (UK)
Alexander Romanovsky, PhD, Newcastle University
Karl King MEng(Hons), MSc(Res) CEng, Frazer Nash Consultancy

SUMMARY

Resilience to human error is a fundamental requirement of any signalling system: signalling interlockings were first deployed to provide resilience against signaller error. The move to relay and then computer based interlockings has progressively expanded the role of interlockings to automate, once manual, actions performed by signallers. As a result of this and the increasing functionality of today's interlockings, resilience against designer error is essential to prevent errors in commissioned interlockings instigating unsafe signalling states. This resilience is becoming increasingly hard to achieve owing to the increasing complexity of computer based interlockings.

Computer science formal methods offers a solution. Already well established for safety critical software development in a range of industries, formal methods are mathematical techniques for automatically proving that complex systems comply with key safety requirements. Whereas earlier attempts to apply formal methods to signalling interlockings have struggled with limited scalability and high upfront costs before benefits can be realised, SafeCap offers an alternative approach. By working within existing signalling design processes and using the highly scalable 'symbolic theorem proving' approach to formal verification, SafeCap provides a demonstrably practical way to realise the benefits of formal verification with minimal upfront costs.

1 INTRODUCTION

Resilience to human error is a fundamental design requirement of any signalling system. However, the continual increase in signalling functionality and complexity makes this requirement ever more difficult (and expensive) to achieve. Now, thanks to cutting-edge computer science, help is at hand. Automated formal verification of even the most complex signalling systems provides a quick and robust defence against such errors.

This paper describes how developments in railway signalling have both provided and eroded the resilience of the railway system to human error. It focuses on the urgent need to ensure increasingly complex and automated computer-based signalling systems are resilient to errors made by their designers whilst still being delivered efficiently. It shows how developments in Computer Science Formal Verification are addressing this need through SafeCap, a collaboration between Newcastle University (UK), SYSTRA Scott Lister and Frazer-Nash Consultancy.

The SafeCap approach differs from earlier applications of formal verification to railway signalling in that it is highly scalable, hence suitable for complex main line layouts, and works alongside existing signalling delivery processes. This means that it can be seamlessly applied as part of a signalling project or retrospectively to legacy data without upfront investment in tools, process development and training. The paper describes practical results achieved through SafeCap, the lessons learnt and planned future developments.

2 RESILIENCE TO ERROR PROVIDED AND ERODED BY SIGNALLING

2.1 The Role of Signalling in Providing Resilience to Operator Error

Interlocking systems on railways perform a range of safety functions, including preventing conflicting moves from being set and ensuring that points are locked in the correct position. They are therefore an essential safety system to the running of the railway. A fundamental role of interlocking systems is to prevent the signaller from setting a conflicting route. This was first achieved by having the levers used to control the points, at that time often located

in local signal boxes, interconnected so that when a route was set by one lever, the other levers were locked mechanically, preventing conflicting routes from being set, hence the term interlocking. The design of interlocking systems then evolved to utilise relays that were hard-wired to create logic circuits that prevented conflicting routes from being set. They have now evolved even further to utilise microprocessor-based electronic systems in what has come to be known as Computer Based Interlocking (CBI) where the logic is written in a software data language. These systems understandably require an intensive amount of design and testing to ensure that they are programmed and implemented correctly so that an unsafe situation does not occur. Therefore, they often require robust safety cases and have stringent requirements on design, testing and commissioning, sometimes requiring specific qualifications specified in law.

Where interlocking systems are provided to mitigate the risk of signaller error, train protection systems are conversely provided to mitigate the risk of driver error. They often require complimentary equipment fitted to both the infrastructure and rolling stock that enables the train to know up to what point it can continue to travel safely based on the current status of the infrastructure. In its simplest form, this was achieved by fitting a lever known as a tripcock to the undercarriage of the trains, which applied the emergency brake when activated. A device known as a trainstop was installed next to the signal, which raised a mechanical arm to activate the tripcock if the signal was set to danger. Modern forms of train protection all work on the same basic principle of stopping a train before a danger point, but utilise various methods, often based around wireless communication, either to stop the train as quickly as possible or to slow it down gradually.

2.2 Vulnerability to Designer Error Brought About by Automation

The evolution of interlockings has brought with it an expansion of functionality that they perform. No longer do interlockings simply provide a defence against human error. They also automate the, once manual, process of setting a signalling route: a couple of button clicks now initiates the movement of any number of points and changes of signal aspect. Furthermore, many additional controls are now incorporated into the interlocking to mitigate specific safety hazards without unduly compromising operational flexibility: swinging overlaps, flank protection, platform signal controls, double reds, axle counter reset and restore controls, etc.

The downside of this automation is a loss of independence. No longer is the agent setting a route (originally a human signaller) separate from the agent checking the safety of that route (originally a mechanical interlocking). They are now one and the same in a modern relay interlocking or CBI, of which the signaller merely *requests* routes. In 1988, the Clapham Rail Crash (UK) provided a stark demonstration of this vulnerability: with no error on the part of the signaller, an interlocking wiring fault caused a train to be authorised at high speed into the back of another train [1]. In 2017, a further collision occurred in similar circumstances at Waterloo station (UK). Underlying each actual accident are multiple wrong-side failures that thankfully did not lead to an accident. How many unidentified errors there are, yet to lead to accidents or wrong-side failures, is unknown.

With greater dependence on the safe behaviour of interlockings, the need for resilience to human error in their design and build processes has long been apparent. Relay circuits and CBI configuration data are meticulously checked by qualified designers, independent checkers and testers to mitigate the risk of errors in the final product. However, whereas it was realistic to verify all possible states of interlocking relay circuits, the same is not true of CBI data. Even for a very simple layout, the number of possible states and paths between those states in a CBI is extremely large. An error anywhere in that data has the potential to cause an accident. As the complexity of CBI data increases so does the problem: the more data there is, the more opportunity there is for a mistake to occur and the harder it is to find it when it does.

The problem is compounded by increasing levels of automation. Not only is route setting automated by the interlocking, increasingly the request to set the route is automatically generated by a signalling control system. This signalling control system itself may be responding to instructions from an automatic traffic management system and the trains no longer controlled by drivers observing signal aspects, but by automatic train operation. The opportunity for an operator to identify an unsafe state and intervene is thus diminishing.

Automation is itself a frequent response to the need for greater railway traffic levels as it enables train movements to be controlled more precisely and quickly. Three concurrent effects thus conspire to magnify the risk associated

with data errors: increasing complexity makes errors more likely; increasing automation removes opportunities for human operators to intervene; increasing traffic levels increase the opportunity for an error to lead to an accident.

2.3 The Impact of Increasing Software Complexity and Control

The general trends in the software industry clearly show us that the complexity of software systems is constantly growing due to the two facts: firstly, software is becoming widely used in new areas because it offers various advantages to the company (ease and reduced cost of modifying, deploying, maintaining, replacing) and secondly, it offers more functionality and flexibility than the traditional technologies used before. This leads to the increased number of software defects in the systems and poses serious challenges to the developers of safety-critical software. There are numerous failures caused by software defects that became famous due their cost and serious effects on the society (including, 1996 Ariane-5, 2007-2009 Toyota unintended acceleration, 2014 Washington state no 911, 2003 Blackout in the US and Canada, and so on). The analysis of the avionics software conducted in 1987 [2] shows that the strategic and tactical software have the defect density of the remaining bugs 9.2 and 7.8 in one thousand lines of software code (KLOS) correspondingly. The lowest known density of the remaining bugs was achieved for the Shuttle software: 0.11 in 1 KLOS with the cost of 1000USD for one line of code of the Primary Avionic System Software [3]. This clearly shows that even with the stringent (and expensive) processes required to achieve the Safety Integrity Level (SIL) 4 certification, the increasing software complexity is likely to lead to the increasing number of errors in safety critical systems.

The area of railway signalling and interlocking clearly follows the trends: this includes the move to software implemented signalling and the growing software complexity to meet the capacity and carbon requirements. The dependence of our society on the railway infrastructure is steadily growing and this, together, with the growing complexity of the software-implemented signalling call for developing novel solutions that help in removing the designer errors earlier in the development process.

Looking into the 60-65 year long history of industrial software production, we could see that in spite of the outstanding successes and a wide proliferation of software into all walks of our life, this industry has always been going through a permanent crisis, with a large number of major projects not completed in time or within budget, often accompanied by the unsatisfactory quality of the resulting products that typically need patching, recalling, or updating. This situation was first realised during the two seminal NATO Conferences in 1968/69 in which term *software engineering* was coined [4]. New solutions promising to solve the problems regularly appear but even when they could improve productivity of the programmers and predictability of the projects, they cannot solve the core immanent problems in building software systems. One may predict that software development will always be in crisis. Unfortunately, as many researchers and engineers believe, software engineering has not become a professional qualification with clearly defined requirements in the way other engineering professions (e.g. signalling or construction) operate. This, combined with the inexorable growth of software complexity (demonstrated by fast advances in ETCS, distributed and heterogenous signalling, autonomous trains, etc.), should probably be considered as a warning for the railway industry in which software is now becoming a critical part of all systems.

3 EFFICIENCY & RESILIENCE IN COMPLEX SOFTWARE SYSTEMS

3.1 Formal Verification

The very first software programs were created by professional mathematicians and invariably accompanied by a formal proof or at least a sketch of argument of their correctness. In fact, a program was merely a practical by-product mechanising a mathematical algorithm. However, the rapid rise in program complexity has rendered traditional hand proof obsolete and nowadays the understanding of a program as an exact mathematical artefact is often but a theoretical possibility. At the same time, the computational power of modern computers enables the construction of automated reasoning tools that can replicate and replace, within a narrow application context, previously manual argumentation of program properties. The kind of reasoning facilitated by such tools is quite different from the kind one may find in a textbook: stringently formal, operating with a limited logical framework and normally rather verbose. It is not unusual, for instance, to have machine proofs with tens of thousands steps yet such proofs are considered simple since they rely on raw computational power rather than intuition and insights

into the nature of a problem. The ability of automated tools to process many thousand proofs without any human input opens the possibility of automated large-scale formal verification that could take us back to programs as mathematical artefacts. And yet powerful automated tools alone are not sufficient. Programs are normally not constructed to be verifiable and often inordinate emphasise is placed on the practicalities of computation often tied to fleeting technological solutions. Hence, while there might well be the potential to verify a program it is often not possible to know what is to be verified, that is, the purpose of a program and its parts, down to the level of individual statements.

The issue of understanding program intent can be remedied by using domain-specific notations and program dialects associated with a well-understood problem domain. In such a setting it becomes feasible to separate the implementation activity of engineer constructing a program and a specification activity defining program meaning and purpose. In a domain-specific notation the latter is fixed once and for all. In a general-purpose notation, like C or Ada, the two activities cannot be separated and must rely on a programmer to supply adequate annotations defining program intent. For this reason, it is very attractive to apply formal verification to safety critical software based on domain-specific notations.

Formal verification of a program seeks to establish that a program satisfies certain semantic properties. Unlike testing, formal verification delivers definite answer of program correctness even the state space or input configurations are extremely large or infinite. In addition, the process of verification would normally yield artefacts justifying the validity of stated results. Static verification, as used in SafeCap, applies automated theorem proving to establish validity of various claims about program properties. A proof script built by a theorem prover can be trivially (in linear time) checked to be a valid proof enabling straightforward reproduction of verification claims. Static verification does not explore individual scenarios or execution case or input vectors and hence it is not directly sensitive to the scale and complexity of program state space.

3.2 The SafeCap Approach

Whilst formal verification has the potential to make the interlocking data design process more efficient and resilient to human error, attempts to apply it frequently encounter barriers of affordability and scalability. Earlier approaches to formal verification have required interlocking functionality to be comprehensively specified in formal notation and verified using highly specialised tools. This requires significant upfront investment to train signalling engineers in formal notation, to re-write signalling principles in this notation and license the necessary software tools. A return on this investment may only be realised years later once several interlockings have been delivered using the new approach. Furthermore, using 'model checking' approaches to formal verification, earlier attempts have struggled to deal with complex railway layouts as the number of model states to verify explodes as complexity increases.

SafeCap overcomes these barriers in two ways. Firstly, SafeCap operates within existing signalling design processes. Interlocking data is read by the SafeCap tool in a format to which signalling engineers are already accustomed; currently SafeCap is configured to read Solid State Interlocking (SSI) Geographic Data Language (GDL), widely used in the UK, though the SafeCap architecture is readily adaptable to other formats. Layouts are entered in the tool as visual signalling plans that signalling engineers are familiar with. Signalling principles are expressed in accessible first order logic, which is readily translated into plain English.

Secondly, SafeCap uses the highly scalable 'symbolic theorem proving' approach to formal verification. This enables the tool to cope with very complex layouts, as may be found in a major railway terminus, as well as simpler ones. It does this by treating the interlocking data as a state transition system in which compliance with signalling principles is verified automatically at each transition step. The size of the verification task thus increases linearly with complexity rather than exploding.

Little upfront investment is needed for SafeCap to deliver benefits. Once SafeCap has been configured to read a specific interlocking language and key signalling principles entered, the tool can begin its work of independently verifying data for compliance with those signalling principles. As an automated tool, verification is very quick and repeatable many times without compromising independence. SafeCap can thus help designers identify errors early on in the design process before data reaches independent checkers and testers. With early deployments of SafeCap, this could save around 5% - 10% of data development costs and reduce delivery timescales by several

months through the avoidance of rework. As the number of signalling principles verified increases, so do these benefits. Crucially, by removing errors before they get to independent check / test, SafeCap provides additional resilience to human error leading to unsafe errors in commissioned signalling systems.

Much like the original concept of an interlocking in mechanical signalling days, SafeCap is there not to act in its own right, but to intervene when a human being attempts to do something unsafe. Whereas a mechanical interlocking intervened by preventing a signaller physically move points / signals in an unsafe manner, SafeCap intervenes by alerting signalling engineers to potentially unsafe aspects of the data that they are producing. In this way SafeCap thus operates as an advisory tool, improving the efficiency and resilience of the signalling design process without replacing it.

3.3 Worked Examples

The SafeCap approach is based on set theory and symbolic transition systems. All the artefacts relevant to a considered problem must be represented as one or the other. Constant sets and relation capture concrete, constant data (e.g., route topology for a specific layout) while a transition system addresses behavioural part.

SafeCap gives a mathematical interpretation to a layout in the terms of an hierarchy of graphs. The track topology is projected to derive track section topology, sub route and sub overlap topology, and, finally, route topology. All these concepts are graphs and interrelated by graph morphisms. Such conceptual, mathematical model of a layout enables reasoning about all the permissible (or safe) actions of an interlocking control is one of three verification ingredients.

SSI GDL is a relatively simple event-based programming notation. Its only non-standard feature worth mentioning is the implied atomicity brackets construct around all code entry points (such as route requests). Atomicity brackets prevent observation of intermediate states within itself so that, for instance, route setting and point command can be done in any order in a route request code block.

We have defined a formal transition systems semantics for SSI and use it to automatically translate SSI source code into a transition system. Since SSI does not have loops and prohibits recursion, the translation is straightforward with the transition system size determined only by the number of code blocks and their cyclometric complexity. As an example, for an interlocking with 174 routes translation of SSI route requests file results in 7,774 state transitions. Such a transition system can be understood as an infinite loop with 7,774 if-then statements inside. Although too large to be legible for a human, its simple structure makes its particularly amenable for symbolic verification.

A third ingredient – a collection of inductive invariants encoding safety principles - relates transition system defining signalling implementation with layout model. Any such principle would define the concrete safety bounds of signalling operation for a given layout. The bounds are then imposed on each state transition to check that signalling respects the bounds. Specifically, a safety principle is applied to each transition system to generate a conjecture. An automatic theorem prover tries to prove each such conjecture and should it fail and error is presumed and reported. It is possible that the prover is simply not powerful and fails to prove a valid conjecture (in addition, the logic we use is undecidable – the cannot be any one single prover to prove all valid conjectures). However, in practice, a combination of per-invariant proof tactics and custom inference rules has proved sufficient in ruling out all false positive cases seen so far.

Formalisation of safety principles as invariants is perhaps the most intricate activity as it requires excellent understanding of the signalling domain, formal methods and the capabilities of underlying verification tools. To facilitate this process, we split the process of definition of a safety principle into three stages: free-form natural language statement written by a signalling engineer, control natural language, also written by a signalling engineer, and a formal statement to be used as an input to the tool. The semi-formal stages aims to resolve ambiguities inherent to natural language statements without using a formal notation and also enforces formulation style suitable to static verification. A formal invariant statement is obtained via manual translation from a semi-formal version.

4 SAFECAP PRACTICAL RESULTS AND FINDINGS

4.1 Scope of Verification Undertaken

To gain a degree of confidence in the tool capability and especially the scalability of the underlying verification principles and tools, we have applied SafeCap to five real-life interlocking layouts.

The example layouts were digitized in SafeCap by a signalling engineer using its built-in editor; signalling data was processed automatically by a plug in translating SSI into the SafeCap formal notation. The plug in is based on our previous on defining formal transitional semantics of SSI.

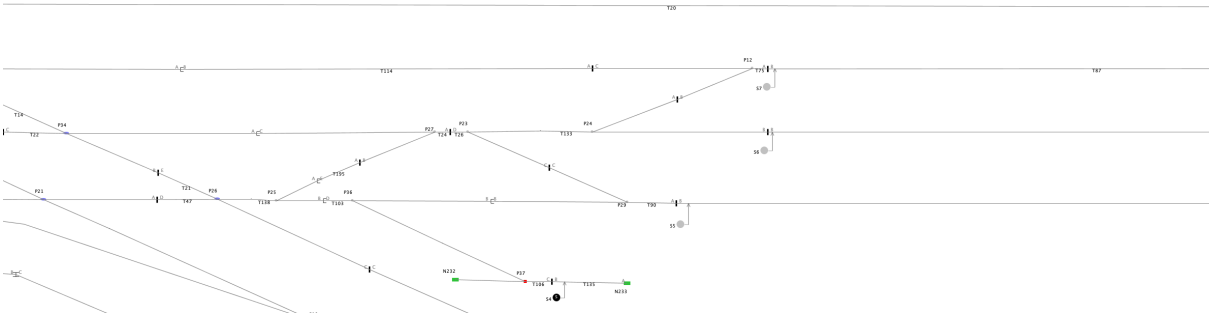

Figure 1. A part of a schema for one of verification cases.

The diagram in Figure 1 depicts a small part of a typical layout of an interlocking under verification. SafeCap uses such diagrams to present counter examples to the verified safety principles.

To state what is to be verified we write safety invariants – predicates collectively defining the safety bounds of a system that must never be violated. In more practical terms, they capture familiar principles of interlocking operations. As an example, consider the principle that on setting a route, all the route sub routes are locked. First, we write a semi-formal statement:

```
[for]
      every exit-controlled route setting command Rxx s
[it holds that]
      every sub route on the route path is locked Uyy l as far as the exit signal
```

The statement must follow one of the predefined syntactic templates, in this case the for/holds that template. We use snippets of SSI syntax to clarify natural language statements. Also, “exit-controlled” is a recognised abbreviation for “a route that has a main aspect controlled or shunt exit signal”. The semi-formal statements is manually translated into a formal predicate. In this case, it uses a set-theoretic form for proof efficiency:

```
route:subrouteset[
      (route_s \ route_s'p) /\
      route:exit~[signal:type~[{ST_Controlled, ST_Shunt}]]] <: subroute_l
```

In blue are the constant relations defined by interlocking layout, in black are SSI variables and terms starting with capital letters are constants.

Verification statements may also be given in a predicate form. As an example, consider the following semi-formal statement:

```
[for]
      every points normal command Pxx cn
[it holds that]
      every sub route over the points in the reverse position is free  Uxx f
```

Its formal translation is:

```
forall p:Node
       point_c(p) != point_c'p(p) and
       point_c(p) == NORMAL
       =>
       subroute:pointreverse~[Node.base~[{p}]] /\ subroute_l == {}
```

So far we have defined 29 safety invariants capturing 21 safety principles (some principles are decomposed into several properties for simplicity). This is not the final set of properties and we are working on building a complete collection with a justification of completeness via a safety case. The following table summarises the range of complexity encountered across the different layouts. Verification times are given for I7-4990K with 32Gb RAM.

Case study	Number of routes	Number of state transitions	Verification time, seconds
N	220	22115[1]	192
T	93	5293	142
O	118	2322	141
PW	56	956	102

4.2 Summary of Findings

During trials on five different layouts, SafeCap has successively identified all known errors in data for those interlockings: two real-world errors, which had already been identified through other processes; twelve errors seeded by the SafeCap team. Errors were deliberately targeted to test that SafeCap could identify violations of a range of signalling principles including

- points deadlocking,
- locking of points within a route,
- opposing route locking,
- sequential route release and
- overlap locking.

As well as genuine errors, the tool successfully picked up instances where signalling principles had omitted for site-specific operational reasons:

- omission of opposing route locking for facing shunt signals in non-passenger sidings;
- omission of opposing route locking to allow a call-on route to be set into an occupied platform in the opposite direction to that in which the first train had entered the platform;
- omission of overlap locking of points for hinge points of swinging overlaps.

Previously unknown issues were also identified that had the potential to lead to unsafe states. These included instances were one interlocking set and cancelled routes, with no signalling principle checks of its own, in response to requests from an adjacent interlocking. SafeCap identified these as violations of signalling principles as it could not prove that the necessary checks were carried out. The situation was not immediately unsafe as long as the checks are carried out before the adjacent interlocking can make the route request. Ensuring that this is the case requires analysis of data on both sides of the interlocking and an understanding of how the interface works. There is therefore a risk of an unsafe state arising should a modification be made to one interlocking without fully understanding the implications on the adjacent interlocking.

SafeCap similarly identified coding practices that, whilst not unsafe in themselves, had the potential to lead to unsafe states should code be modified without fully understanding its original design. For example, SafeCap identified situations where sets of points could be commanded to move without explicit tests that route locking over those points was free. Subsequent analysis identified that these violations of safety properties could only occur if a different set of points was commanded in a position contrary to the direction of a sub-overlap over those points.

[1] In four separate interlockings.

Therefore, as long as the latter never occurred neither did the former unsafe state. It was relatively straightforward to modify SafeCap safety properties to take account of this and successfully verify that the route locking principle was upheld. It is much less straightforward for a human being to follow such complex logic and avoid unwittingly introducing errors when modifying the code as a result.

4.3 Lessons Learnt for Interlocking Data

As well as offering additional resilience to designer error in its own right, SafeCap has also provided insights into how resilience could be incorporated into the way data is structured.

Simplicity. The logical arguments by which safety is assured in real-world interlocking data can be very complex and therefore hard to follow. A real-world example: a set of points is regarded as locked by route locking, because the only situation in which the points can move when the locking is free is when a sub-overlap somewhere else is locked in a direction contrary to the commanded position of points under a subsequent sub-overlap. An explicit test of route locking before each command to move the points would be much easier to follow and verify, potentially reducing the risk of error.

Whilst complexity may once have been necessary to make efficient use of scarce processing power, it is less so nowadays. Ironically, the abundance of today's computing power has actually led to greater complexity in order to accommodate new safety and operational requirements. With more complexity comes more opportunity for error, even with SIL certified processes. Consequently, the appropriateness of placing functionality that could be performed elsewhere in a high-integrity interlocking needs to be challenged. Where the safety benefit of an additional control is low, it might be safer to implement it in a lower integrity device rather than risk introducing errors into core interlocking functionality.

Defensive programming. Signalling principles are not always explicitly verified in interlocking data, but often implicitly on the understanding that certain states cannot occur. For example, in the UK, opposing route locking is generally only verified explicitly for the last sub-route in an opposing route. Other opposing sub-routes are verified implicitly on the basis that they are locked at the same time as and never released before the last sub-route.

Such approaches have the advantages of efficiency and, following an 'offensive programming' philosophy, making some errors easier to detect. However, efficient use of computing power is less of a concern now and not all errors do manifest themselves easily. There may, therefore, may be merit in adopting more defence approaches to writing interlocking data whereby signalling principles are verified more explicitly. As well as making the data easier to verify, such a 'defensive programming' approach could provide greater resilience to individual errors.

Modularisation. Modern CBIs typically perform a wide range of functions: some associated with set routes, some enforcing safety constrains, some providing additional controls to address specific safety hazards. Where functions are written together in the same configuration data, there is a risk of an error in one part of the code causing a safety hazard somewhere completely different. For example, a typographical error in the naming of a set of points required for flank protection (mitigating a safety hazard) could result in another set of points moving underneath a train. Resilience to such errors could be achieved by braking the data up into small, well defined modules.

The concept of structuring complex software as discrete modules is already well established through object oriented programming. Arguably it was actually pioneered by railway signalling engineers in the geographic interlockings of the 1960s and -70s. As an approach, it enables complexity to be managed more effectively: each object is highly constrained in what it can do, the data it has available to it and its interfaces to other objects. For example, a hypothetical 'points' object may perform deadlocking functionality and allow the points to be locked and unlocked by 'route' objects, but little else. With small, highly constrained modules, verifying compliance with safety constraints becomes more straightforward.

5 FUTURE DEVELOPMENTS

SafeCap has proven its ability to identify errors in CBI data developed for existing signalling installations. The next stage of development is to utilise the SafeCap tool to test data during the design phase as a secondary check of the usual design process to assist with identifying data faults. This will prove the tool's pedigree for identifying

errors in the data of a real application. During this trial an initial Hazard Identification for the tool shall be carried out in order to map out the product approval strategy.

The next stage will be to provide the SafeCap tool to test data during the design phase as the main formal method of testing the data and identifying faults on a trial CBI upgrade project. During this trial, a Generic Application Safety Case (GASC) will be developed to achieve product acceptance for SafeCap to be used as a design tool for the product line used on the trial project (SSI, WESTLOCK, SMARTLOCK, etc.) in line with EN 50128.

Ultimately SafeCap will be utilised to test data during the design phase as the main formal method of testing the data and identifying faults on CBI upgrade projects. The SafeCap GASC will be incrementally upgraded for product acceptance whenever the tool is used for a new interlocking system.

6 REFERENCES

1. Stacy M., 25 years after Clapham – A review of signalling procedures, Rail Engineer, https://www.railengineer.co.uk/2014/01/06/25-years-after-clapham-a-review-of-signalling-procedures/ (accessed 12th April, 2019).
2. J. McCall, W. Randall, et al. Methodology for Software Reliability Prediction Rome Air Development Center. USA. Final Technical Report RADC-TR-87-171, Volume II. 1987
3. Jenkins D., Advanced Vehicle Automation and Computers Aboard the Shuttle, NASA, https://history.nasa.gov/sts25th/pages/computer.html (accessed 12th April, 2019).
4. Randell B., The 1968/69 NATO Software Engineering Reports, http://homepages.cs.ncl.ac.uk/brian.randell/NATO/NATOReports/ (accessed 13th May, 2019).