# Building Functional Safety into Complex Software Systems, Part II

Chris Hobbs, Kernel Developer
QNX Software Systems
chobbs@qnx.com

## Abstract

Traditionally, demonstrations that software systems meet standards for functional safety have depended on exhaustive testing. This method is adequate for relatively simple, deterministic systems, with single-threaded, run-to-completion processes. It is inadequate, however, for today's multi-threaded systems. The complexity of these systems precludes their being treated as deterministic systems *in practice*.

In Part I of this whitepaper series we discussed the limits of testing of complex software systems, and some factors that should be weighed when deciding how to build complex software systems that must meet functional safety standards.

In Part II, we propose how a combination of procedural rigor, statistical testing, and design verification can be used to increase confidence in the functional safety of complex software systems. In subsequent papers in this series, we will explore specific strategies for building and validating functional safety in complex software systems.

## Preparing Functional Safety

Achieving functional safety for a software-based system is not trivial; it is a long and costly process that should not be undertaken lightly, but one that can bring in enormous returns.

As with any important project, the first order of business should be to assemble a team of experts who will set down a clear statement of precisely what will be claimed for functional safety and in what context, and what evidence will be presented as proof. The requirements that follow from this statement should become an integral part of the project right from the start.

Five key tasks should be addressed; the first two must be addressed at the start of the project, for they define the requirements for the other tasks. The five tasks are:

- define sufficient dependability
- establish good development processes
- establish methods of validation
- build functional safety
- validate the functional safety of the system

## Defining *Sufficient* Dependability

A system's *dependability* is its ability to respond correctly to events in a timely manner, for as long as required; that is, it is a combination of the system's *availability* (how often the system responds to requests in a timely manner), and its *reliability*

(how often these responses are correct). In other words, a dependable system is a system that responds when it is required in the time required, and responds correctly.

When planning functional safety, we must define precisely the criteria against which the system's dependability is to be measured. This means staying clear of facile marketing claims of the five-nines sort: available 99.999% of the time; *ergo* completely dependable except for five minutes 16 seconds of the year. This type of claim is meaningless unless more information is offered about how this time when the system is not dependable is distributed throughout the year.

If this claim is made about, for instance, a flight control system on an airliner it has very different implications if the five minutes 16 seconds (0.001% failure) occurs all at once, or if it is spread across one million distinct instances of 316 microseconds (also 0.001% failure). See Table 1 below for examples of various possible meanings of the phrase "five-nines availability".

Five minutes, 16 seconds can mean a catastrophic failure, while one million distinct instances of 316 microseconds may have no effect on the system's dependability and may even go completely unnoticed. In fact, the airliner flight control system may well tolerate a system that guarantees only four-nines availability (99.99%), if unavailability is distributed over one million instances of 3.16 milliseconds per year separated by sufficiently long instances of availability. It is also worth noting the duty cycle of the software. Few flight control systems run for more than 20 hours at a stretch, after which they can be restarted, forcing rejuvenation.

| Failures per year | Duration of each failure | |
|---|---|---|
| 1 | 5 minutes 16 seconds | Potentially catastrophic |
| 10 | 32 seconds | |
| 100 | 3.2 seconds | |
| 1000 | 316 milliseconds | |
| 10,000 | 32 milliseconds | |
| 100,000 | 3.2 milliseconds | |
| 1,000,000 | 316 microseconds | Possibly benign |

*Table 1. Five-nines availability as it might affect a flight control system.*

Thus, a careful and comprehensive definition of dependability requirements serves a dual purpose. First, it provides an accurate measure against which a system's functional safety can be validated. Second, by clarifying what is indeed functionally required, it eliminates vague (and therefore meaningless) requirements, and removes from the project bill the effort and cost of trying to meet them.

## Establishing Good Processes

A good process does not guarantee that the system being built with that process will achieve the required level of functional safety. It does not even guarantee that the system will be a good one.

With a poor process it is possible to build a good system or even one that meets functional safety requirements, but to do so is much more difficult than with a good process. Further, a good process provides a well-defined context within which test results can be interpreted. By setting limits around what is tested it sets the scope of the tests and, by inference, the scope of what must be validated by methods other than testing. With a clear understanding of what will be verified by what method, it becomes possible to make clear and substantiated claims about a system's conformance to functional safety standards.

For example, if we return to the five-nines claims discussed above, it is quite likely impossible to prove by testing that the system is dependable all but a maximum of 316 seconds a year, because to do this we would have to run the system under real load conditions for many years. It may be possible, though, to show by other means that the system recovers from errors within 316 milliseconds, and that this is acceptable because in the context where the system is used (an airliner rather than, say, a jet fighter) any error under 500 milliseconds has no appreciable effect on control of the aircraft.

## Establishing Methods of Validation

A standard requirement for certification of a system's safety integrity level (SIL) is that the system's functional safety characteristics be measured, validated and demonstrated. This validation involves explicit claims, evidence, and expertise, as identified in *Software for Dependable Systems*, edited by Daniel Jackson.

### Explicit claims

No system can be absolutely dependable. Hence, it is essential that any and all claims about the system's dependability be clearly and explicitly articulated; that is, what "sufficiently dependable" means for this system.

### Evidence

Evidence supports the assertion that the system meets the requirements for sufficient dependability, as stated in the explicit claims. This is, of course, what everyone — accreditation bodies, auditors, and customers — will be looking for. Just as no system is absolutely dependable, no method of validation is absolutely fool-proof. This uncomfortable truth is particularly relevant to the validation of complex systems. Hence, validation of such systems will include evidence obtained by a variety of different methods.

Standards such as IEC 61508 and EN 50128 list a number of viable validation methods. "Validating Functional Safety" on page 4 below offers some comments about how functional safety can be validated in a complex software system.

### Expertise

Just as failures must at times be attributed to human error, the absence of failures should often be attributed to human expertise. Ultimately, it is the relevant experts (system architects, software designers, process specialists, programmers, verification specialists, etc.) who set the requirements for a sufficiently dependable system, who build the system, and who validate that it meets its requirements.

Extensive knowledge of both context and problem is needed to formulate and justify requirements. No two software designs are alike, and great expertise is required to evaluate different solutions and select the designs that best meet these requirements.

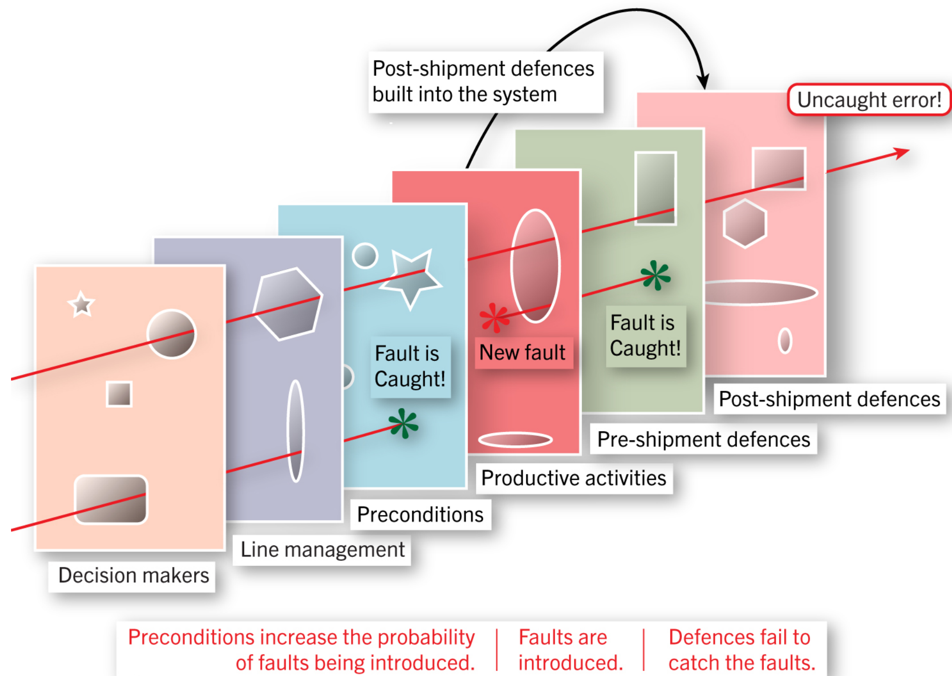


*Figure 1. Reason's model (adapted) of how faults become failures applied to software design and development.*

Finally, a comprehensive understanding of software validation methods, the software system being evaluated, and the context in which it is evaluated (including validations of similar systems) is required to demonstrate that the software system in question meets its defined functional safety requirements.

## Building Functional Safety

Validation demonstrates only that a system does (or does not) meet defined dependability standards. Functional safety must be build into a system from the start, and all work to this end should follow from the premise that *all software contains faults and these faults may lead to failures.*

### From fault to failure

Not all faults become failures, which is fortunate because every stage of system development, from initial conception, through design and development to deployment, incorporates imperfections. Failures are the product of a series of conditions, decisions and actions.

Figure 1 above shows James Reason's model of how faults become failures, in which we have subdivided the defences to match the two layers in software: pre- and post-shipment defences (with attendant holes). Pre-shipment defenses are those validation and verification activities carried out before deployment, while post-shipment defenses are defenses built into the system itself and activated to protect it during use. The causes of every failure can be traced back—at least in theory—to a lacuna at each stage.

| | |
|---|---|
| Fault | A mistake in the code, which may or may not cause undesired behavior. |
| Error | Undesired behavior caused by a fault in the code. |
| Failure | A system failure caused by an uncontained error. |

*Table 2. Faults, errors and failures*

For example, a decision maker may decide that software should be written in a language, such as C, that doesn't provide much protection against programmer errors. Line management may organize work around teams in ways that do not foster good coding practices. Developers may be working under conditions that lead them to make mistakes: an inadequate tool set, overly-aggressive deadlines, lack of sleep, etc. Designers and developers are human, and therefore produce flawed designs and code. Testing and design verification misses some of these flaws, so they are not corrected. Finally, post-shipment defences, such as code written to recover from errors, may fail.

Each stage of the process introduces faults. Some of these imperfections are benign, others are caught and corrected, or at least prevented from causing errors, others cause errors, and unfortunately, some of these errors cause failures. For example, a tired or just clumsy developer may want to allocate 10 bytes of memory, but may type:

```
char  fred[100];
int   x;
```

or:

```
char  fred[1];
int   x;
```

Both are faults, since they do not allocate the correct number of bytes. In the first case, unless the system is subject to severe memory restrictions, the fault is unlikely to produce an error, much less a failure. The second case, however, may well produce an error, since the programmer, who believes he has a 10-byte buffer, may write code that will overwrite not only **x**, but whatever is in the next five bytes as well. This error could, of course, cause a failure.

**Heisenbugs**

Heisenbugs may manifest themselves anywhere in a system. They are a particular threat to mission- and safety-critical systems — time-bombs known to be in the application, but impossible to track down or remove.

## Multiple lines of defense

Working from the premise that all faults may lead to failures, we must include multiple lines of defense when we design and build a functionally safe system:

### Isolate safety-critical processes

Identify safety-critical components and processes, and design the system so that they are isolated and cannot be compromised by other components or processes. An automobile digital instrument cluster, for instance, is always isolated from vehicle infotainment systems, though they may both share some display space on the dashboard.

### Reduce faults

Know and understand the environment in which the system will operate, use suitable tools, and design and build for this environment. Staff the project with people with the necessary expertise and experience, people who will be able to select, design and build the best possible system within the context of the project's sufficient dependability requirements. For example, follow best practices (for a start, check return values or catch exceptions!), and ensure that working conditions allow designers and developers to do their best work.

### Prevent faults from becoming errors

Design and build the system so that faults do not become errors. While the ideal solution is to identify and remove faults from the code, it is crucial to always assume that some faults will remain undetected. The systems should be designed so that as much as possible, faults are encapsulated and do not become errors once the system is deployed.

For example, if we return to the simple example of the elevator system described in Part I of this series[1], we could put a timer and a trip counter to force the elevator to stop and open its doors at some floor after a specified time has elapsed, or after it has made more than a specified number of trips without opening its doors. This mechanism would ensure that a fault like the one in our example (which allows the elevator to continue moving between floors forever without stopping and opening its doors) did not become an error and, from that, a failure.

### Prevent errors from becoming failures

Again, assuming that no software is fault-free and that faults will at some point cause errors, design to prevent errors from becoming failures. Replication is a common method for preventing errors from becoming faults. It is used extensively with hardware, but can also be useful for software.

Various replication models are possible, including transactional replication, where a passive system is synchronized with the currently active system and takes over in the event of a first system failure; and group synchrony, where unsynchronized systems all perform all requested tasks and the requester (or consumer, or client) accepts either the first result, or all results and uses the result of majority consensus.

Specific for software, data and information can be replicated by being stored in different locations and different formats, or provided with error-detection and error-correcting bits to guard against loss and corruption. Time (or processing) replication, which performs the same computation multiple times, either on the same or different processors, can help avoid availability issues, particularly those associated with Heisenbugs.

We will examine in more detail specific techniques for building and validating functional safety in subsequent instalments of this whitepaper series.

## Validating Functional Safety

After taking all possible care to ensure that functional safety is designed and built into a  software system (that, for instance, deadlines and other pressure did not cause anyone to make mistakes or deliver shoddy work) the system is not functionally safe until it is *proven* functionally safe. Claims about its dependability

[1] Chris Hobbs *et al,* "Building Functional Safety into Complex Software Systems,  Part I". QNX Software Systems, 2011.

must be validated. For a complex software system, this validation must comprise at least testing and design verification.

## Testing

Testing asks the question: "What confidence can we have in the assertion that the module, integration unit or system has correctly implemented its requirements?" Figure 3 on the right illustrates how, in a traditional waterfall development, the various levels of testing correspond to the levels of architecture and design.

In any but the simplest system, as one moves up the right-hand part of the V (the grey box in Figure 3), testing inevitably becomes less and less thorough, and the results we obtain from testing must be understood as a sampling, which we must analyze statistically.

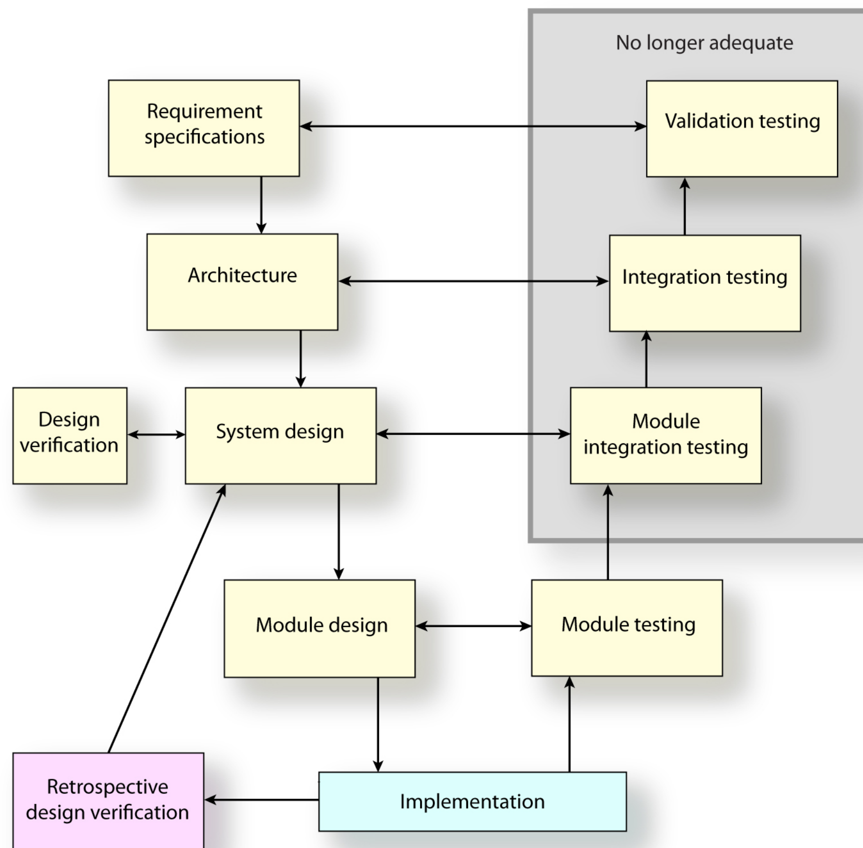


*Figure 2. Testing, statistical analysis, and retrospective design validation complement each other to validate the functional safety of a complex software system.*

The exception to this rule is module testing, which by its nature is not affected by the temporal complexities of a system or subsystem. Here, techniques such as symbolic execution can complement manual test case generation, using means such as equivalence class partitioning and boundary value analysis. A tool such as the KLEE Symbolic Virtual Machine can automatically generate test cases with good path coverage that can form the foundation for more focused, manually-generated tests.

However, for the reasons outlined in Part I of this series, with today's complex software systems such certainty is no longer possible in the test layers above module testing. We cannot provide a statistical "cookbook" for the analysis of test results, and testing must be undertaken by with techniques like those described below.

## Statistical analysis of random tests

Statistical analysis of random tests attempts to determine the following: "If $N$ randomly-chosen tests have been run on the system without a failure being invoked, then what confidence can we have that the system will meet a particular failure rate? This is normally calculated as

$$M = 1 - \left(1 - \frac{1}{h}\right)^{N}$$

where $M$ is the confidence, $h$ is the acceptable failure level (e.g. 1000) and $N$ is the number of tests.

This formula alone illustrates the effort involved in testing. For example, if the requirement is that the system meet IEC 61508, SIL3 (Safety Integrity Level 3) for a low demand system; that is, if it must operate correctly 9,999 times out of 10,000, then $h$ = 10,000. If we require a 99.5% confidence that our system meets SIL3 ($M$ = 0.995), then the number of tests required is

$$N = \frac{\ln(1 - 0.995)}{\ln\left(1 - \frac{1}{10000}\right)} \approx 53{,}000$$

In other words, for the safety analyst (and the auditor) to have a 99.5% confidence that the low-demand system meets IEC 61508 SIL3 standards, we would need to execute over 50,000 random tests representative of the field operation of the system without detecting a failure. Should we encounter a failure, we would need to correct the system and run a further set of 50,000 random but representative tests.

The main problem with this technique is the difficulty we have in producing random tests that are representative of the use to which the system will be put. If our formulation of the system's dependability claims includes a good definition of the environment and what is meant by "sufficient dependability" for that system, we may be able to reduce the number and scope of these tests

For instance, if we determine that printing is not required for sufficient dependability, we may be able remove printing from our claims and thus reduce our test cases—assuming, of course, that we can demonstrate that we have isolated printing in its own address space and that it cannot adversely affect anything else in the system. We will not, however, be able to make the problem go away, and will need to use other techniques to help demonstrate our system's sufficient dependability.

## Remaining fault estimation

Over the years many functions have been applied to estimating the number of remaining faults and thereby determining the point at which testing is no longer economical. These functions are particularly difficult to determine because, as is well known, correcting one bug often introduces others. Fault injection is a statistical technique for determining, at any point in the testing cycle, the number of significant faults that remain in the system.

With fault injection, bugs with characteristics of possible remaining (unknown) bugs are deliberately introduced into the code. This deliberate introduction of bugs in the code has several beneficial effects.

- Most system testing does not exercise the code that is executed if a fault causes an error and recovery occurs. When we deliberately inject faults into the main system, we cause this detection and recovery code to be invoked and thereby tested.
- By their nature, the types of non-reproducible, timing-related bugs that are the bane of complex systems are not normally isolated during conventional system testing. By deliberately introducing faults and determining what percentage of these faults are identified by the testing, we can estimate the total number of inadvertently introduced faults remaining in the code by assuming that the distribution of seeded to unknown bugs in the systems is the same as the distribution in the test. With this information we can determine what and how much further testing (and correction) the system requires in order to meet its functional safety requirements.

**The de Havilland Comet Catastrophes**

The de Havilland Comet airliner catastrophes provide excellent examples of an inadequate definition of the environment in which functional safety was required.

Metal fatigue, specifically unforeseen stresses around the airliners' rectangular windows, caused two Comets to disintegrate over the Mediterranean in January and April 1954. These failures occurred despite the Comet's having been subjected to testing in a water tank and decompression chamber for the equivalent of 40,000 hours of service. No one at the time understood the kinds of stresses to which a jet airliner is subjected — no one understood the environment.

The chief difficulty with fault injection is creating artificial faults with the same characteristics as the (unknown) faults already in the software. Injected software faults can randomly corrupt data (simulating a race hazard), a program (difficult in systems with good memory protection) or timing (by artificially consuming processor cycles and preventing other programs from running). Certainly, as with statistical analysis of random tests, the results of this type of fault injection require careful statistical analysis.

We hope that, if we have made anything clear in this brief discussion on validating functional safety, it is not just that testing alone is inadequate to validate functional safety, but especially that no one method is sufficient. Testing and statistical analysis are part of the answer; they should be complemented by other methods of validation, such as design verification.

## Conclusion

We saw in Part I that the functional safety of today's multi-threaded complex software systems cannot be validated by traditional, state-based testing alone. Though these systems are deterministic in theory, due to the number of possible states and state transitions they can present, they might as well be infinite. It is, nonetheless, not only necessary, but possible to build functionally safe complex software systems.

In Part II we have seen that functional safety should be designed and built into a software system from its inception. It begins with the best available expertise and a clear definition of the system's dependability requirements: its *sufficient* dependability. This definition of what is meant by sufficient dependability must be

followed by the application of rigorous standards and practices throughout the system design and development, and a comprehensive validation program that includes, in addition to traditional state-based testing at the module level, statistical testing, and design verification. We will address design verification, and, more specifically, *retrospective* design verification, in Part III of this series.

## References


*Agency Risk Management Procedural Requirements* (NP4 8000.4A). NASA, 16 Dec. 2008.

Berard, B. *et al*. *Systems and Software Verification*. Berlin: Springer, 2001.

Bouissoe, Marc, and Fabrice Martin and Alain Ourghanlian. (1999) “Assessment of a Safety-Critical System Including Software: A Bayesian Belief Network for Evidence Sources”. *Proceedings of the Annual Reliability and Maintainability Symposium*.

*EN 50126 1999: Railway applications — The specification and demonstration of reliability, availability, maintainability and safety* (incorporating corrigenda May 2006 and May 2010).

ERA Technology Ltd. (2009) “Risk Modelling using Bayesian Networks”. http://www.era.co.uk

Havelund, Klaus *et al*. “Formal Analysis of a Space Craft Controller Using SPIN”. Moffet Field: NASA Ames Research Center, n.d.

Helminen, Atte. (2001) *Reliability estimation of safety-critical software-based systems using Bayesian networks*. Helsinki: Säteilyturvakeskus (Finnish Radiation and Nuclear Safety Authority). http://www.stuk.fi/julkaisut/tr/stuk-yto-tr178.pdf

Hobbs, Chris. “Fault Tree Analysis with Bayesian Belief Networks for Safety-Critical Software”. QNX Software Systems, 2009. www.qnx.com.

_____. “Protecting Applications Against Heisenbugs”. QNX Software Systems, 2010. www.qnx.com.

Hobbs, Chris, e*t al*. “Building Functional Safety into Complex Software Systems, Part I”. QNX Software Systems, 2011. www.qnx.com.

Jackson, Daniel, ed. *Software for Dependable Systems: Sufficient Evidence?* Washington: National Acadamies Press, 2007.

Lions, J. L. *et al. Ariane 501 Inquiry Board Report*. Paris: ESA, 1996.

Littlewood, Bev and Peter Popov, and Lorenzo Strigini. (2001) “Modeling software design diversity — a review”. *ACM Comput. Surv*., 33(2):177-208.

*QNX® Neutrino® RTOS Safe Kernel 1.0: Safety Manual: QMS0054 1.0.* QNX Software Systems, 2010. www.qnx.com.

*Railway Safety: Engineering Safety Management Yellow Book 3: Application Note 2: Software and EN 50128*. Issue 1.0. London: Railway Safety, 2003.

Reason, James. *Human Error*. Cambridge: Cambridge UP, 1990.

## About QNX Software Systems

QNX Software Systems Limited, a subsidiary of Research In Motion Limited (RIM) (NASDAQ:RIMM; TSX:RIM), is a leading vendor of operating systems, development tools, and professional services for connected embedded systems. Global leaders such as Audi, Cisco, General Electric, Lockheed Martin, and Siemens depend on QNX technology for vehicle infotainment units, network routers, medical devices, industrial automation systems, security and defense systems, and other mission- or life-critical applications. Founded in 1980, QNX Software Systems Limited is headquartered in Ottawa, Canada; its products are distributed in more than 100 countries worldwide. Visit www.qnx.com and facebook.com/QNXSoftwareSystems, and follow @QNX_News on Twitter. For more information on the company's automotive work, visit qnxauto.blogspot.com and follow @QNX_Auto.

### www.qnx.com

© 2011 QNX Software Systems Limited. QNX, QNX CAR, Momentics, Neutrino, Aviage are trademarks of QNX Software Systems Limited, which are registered trademarks and/or used in certain jurisdictions. All other trademarks belong to their respective owners.
302191 MC411.86