Protecting Software Components from Interference in an ISO 26262 System

Building Functional Safety into Complex Software Systems, Part IV

Chris Hobbs, Senior Developer, Safe Systems
Yi Zheng, Product Manager, Safe and Secure Systems
QNX Software Systems Limited

Automobile safety often depends on the correct operation of software-based systems constructed from many different components. Good design requires that these components be isolated from each other on multiple axes so that they do not inadvertently interfere with each other.

I*SO 26262: Road vehicles—Functional Safety*[1] provides explicit guidance concerning interference, drawing particular attention to risks associated with interference between components of different Automotive Safety Integrity Levels (ASILs). It describes mechanisms to reduce these risks, suggests how partitioning must be handled, and specifies how the ASILs of composite systems must be calculated. In particular, paragraphs 7.4.11 and 7.4.12 of Part 6 (which refer to annex D) prescribe the manner in which partitioning must be handled, and paragraphs 6 and 7 of Part 9 specify how the ASILs of composite sub-systems are to be calculated.

Figure 1.A digital instrument cluster showing information from components requiring different dependability levels

In this paper we present techniques that can help a) ensure that a system implements the component isolation required by ISO 26262, and b) demonstrate that this isolation has been implemented. We make no claims to offering anything more than an incomplete catalogue of some important techniques worth considering. Adequate treatment of each technique would require a book-length study, if not a library of studies. For more information, please contact us at chobbs@qnx.com or yzheng@qnx.com.

[1] First edition, 2011.

Interference

Many systems built in accordance with ISO 26262 contain software elements developed to different ASILs. The incorrect behavior of one of these components could potentially interfere with the behavior of another component and lead to a violation of safety requirements.

A violation of safety requirements occurs, not just when a lower-ASIL component interferes with a higher-ASIL one, but also when two components with the same ASIL interfere with one another—or even when a higher-ASIL component interferes with a lower-ASIL component. To help manufacturers avoid developing all components to the standard of the highest ASIL in the system, ISO 26262-6, paragraph 7.4.10 permits the use of software partitioning, in accordance with ISO 26262-9, paragraph 6, which defines interference as "the presence of cascading failures from a sub-element with no ASIL assigned, or a lower ASIL assigned, to a sub-element with a higher ASIL assigned leading to the violation of a safety requirement of the element".

Types of interference

Interference comes in many types and differs according to whether the components are actively cooperating or are meant to be completely independent. The following is an incomplete list. One component may:

- rob another component of system resources (file descriptors, mutexes, flash memory, etc.). By periodically using and not releasing a file descriptor, one process could eventually consume all the system's file descriptors and prevent a crucial process from opening a file in the flash memory when it needs to.
- rob another component of processing time, preventing it from completing its tasks. By performing a processor-intensive calculation or by entering a tight loop under a failure condition, a process could prevent a critical process from running when it needs to.
- access the private memory of another component. In the case of read access, this may be a security breach that could lead to a safety problem later; in the case of write access, this could immediately create a dangerous situation.
- corrupt data shared with another component, causing the other component to behave in an unexpected and potentially unsafe manner.
- create a deadlock or livelock with another component with which it is cooperating. In either case, the system makes no forward progress, allowing a dangerous situation to arise through inaction. The circumstances that give rise to deadlocks and livelocks are generally subtle and, because of their temporal nature, can seldom be detected or reproduced by testing.
- break its contract with a cooperating component by "babbling" (sending messages at a high rate or repeating messages) or sending messages with incorrect data.

The isolation axes

Isolation is a fundamental strategy for protecting against interference. No single form of isolation is sufficient. Table 1 below provides an overview of isolation techniques, which are described in more detail below, as are techniques for ensuring isolation and for avoiding deadlocks and livelocks. It offers a summary of the four isolation axes shown in Figure 2 below.

Isolation	Description	Limitations
Spatial	Fundamental to all other forms of isolation. Provided by a memory-management unit (MMU) that protects the memory of each process from being accessed (read or written) by any other process.	Not sufficient in itself. Hidden, incorrect assumptions in, for instance, formal design checking[2] can invalidate safety claims supported by proofs generated by these techniques.
Temporal	Rate- and deadline-monotonic scheduling can be used to demonstrate that processes in a simple system won't be starved of processing time and will meet their real-time deadlines.	Relatively inflexible. Difficult to incorporate varying overheads, aperiodic tasks, caching, varying priorities, bursty events, multicore processors, intrinsically unpredictable hardware (for instance, buses relying on collision detection). May require discrete event simulation[3] and statistical guarantees to show that processes meet real-time obligations.
Data	Static (unchanging) data: the memory pages containing the data are normally marked as read-only and a checksum can be used to ensure data integrity. Dynamic data: replication and diversification (multiple instances of the same data stored in semantically different forms).	Performance penalty for verifying checksums before each access or for replicating data may be unacceptable.

[2] See Michael Fisher, *An Introduction to Practical Formal Methods Using Temporal Logic*, Chichester, U.K.: Wiley, 2011.

[3] See many discussions, including in Jack P.C. Kleijnen, *Design and Analysis of Simulation Experiments*, New York, Springer, 2008, and many other texts.

Isolation	Description	Limitations
Resource	`rlimit` parameters can prevent one process from starving others. An anomaly detection program can learn what constitutes normal behavior, monitor resource allocations, and take corrective action.	`rlimit` may fail to catch a process writing a large number of small files to flash memory. Often difficult to find training sets for supervised learning programs. An anomalous situation can occur very quickly and cause damage before it is detected.

Table 1. Overview of the isolation axes

Isolation techniques

In general, it is best to isolate as many components as possible, using a variety of complementary techniques.

OS architecture

In a microkernel OS, components (file-system, device drivers, network stack, etc.) run in their own address spaces, isolated from each other as well as from the kernel.

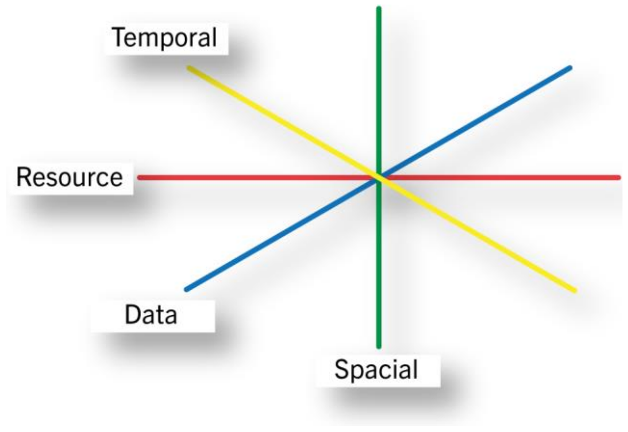

Figure 2. The isolation axes

Rate- and deadline-monotonic scheduling

Rate- and deadline-monotonic both refer to scheduling policies and tools that can be used in simple systems to ensure that processes will meet their real-time deadlines.

In a real-time system it is incorrect to assign priorities according to a process's "importance". Other methods must be used. With rate-monotonic scheduling, for example, the processes with greater execution rates receive the highest priorities[4]. That real-time deadlines will be met can then be mathematically proven for some systems.

[4] Briand, Loïc and Daniel Roy Meeting Deadlines in Hard Real-Time Systems: The Rate Monotonic Approach, IEEE Computer Society, 3rd ed., 1999.

Adaptive partitioning

Adaptive time partitioning can ensure that processes are not starved of CPU cycles, while also making sure that system resources are not wasted. It assigns minimum levels of processor time to a group of threads to use *if the threads need it* (see Figure 4). This technique reduces the work required to ensure that processes aren't starved of cycles, and can be applied in complex systems where rate- or deadline-monotonic scheduling cannot be used.

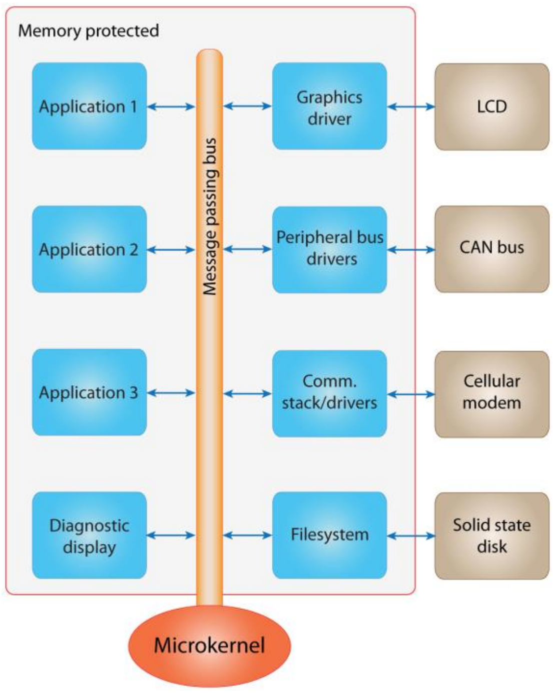

Figure 3.A microkernel OS architecture

Effective use of adaptive partitioning

Note that to make effective use of adaptive partitioning in a multi-core system, it is especially important to ensure that *all* available processor time is considered when time is being shared between the partitions.

If all threads in a particular partition are waiting for external events, then no time needs to be allocated to that partition. On the other hand, if a) the threads in partition X could use more than their allotted time, b) these threads have a priority higher than other ready threads, and c) processing time is available, then the threads in partition X can be allocated the available time and executed.

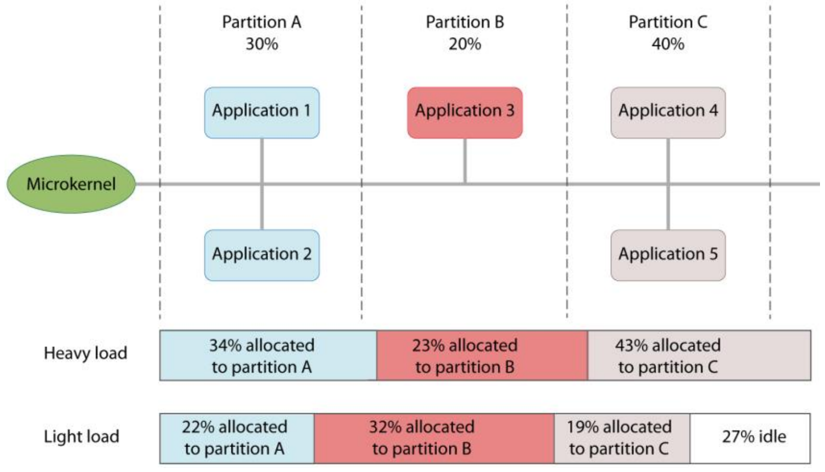
Partition A
30%
Partition B
20%
Partition C
40%
Application 1
Application 3
Application 4
Microkernel
Application 2
Application 5
Heavy load
34% allocated to partition A
23% allocated to partition B
43% allocated to partition C
Light load
22% allocated to partition A
32% allocated to partition B
19% allocated to partition C
27% idle

Figure 4.An example of adaptive partitioning.

Debugging thread

Adaptive partitioning simplifies the analysis required to ensure that all critical threads meet their real-time budgets. In many systems it may also be useful to allocate a debug thread to a partition, and guarantee this thread processing time if it needs it.

Under normal conditions, this debug thread is idle and, therefore, generating no overhead. However, if a high-priority process begins to run continuously, locking up the system, if the debug process has a sufficiently high priority it can acquire the cycles it needs to gather diagnostic information and interact with the programmer.

Data diversification

Data diversification is the storing of data in different semantic ways.[5] It provides an additional level of confidence in the integrity of the data, based on the premise that, while one algorithm may give a wrong result, it is unlikely that two different algorithms will give the same wrong result.

For example, the distance that a car has travelled since a particular event occurred might be stored as a) the number of rotations of the wheel and b) an integration of the car's speed at regular intervals. The information (distance travelled) is the same in both instances, but with the information stored in two formats the system can check for discrepancies, possibly, discover an algorithmic error in one of the calculations used to produce the information—and, perhaps, even correct it.

There is also, of course, the possibility of data corruption, not because a process fails, but because of errors in the memory device.[6] Such corruption is normally caught by using ECC memory devices, and is not a component isolation problem.

Anomaly detection

Predictor-corrector methods such as Kalman filters have traditionally been used to detect anomalies in external sensor inputs. These methods are less well suited for detecting anomalies in system variables, where there is rarely a random noise on a value. However, in a multi-threaded system random noise is usually present, so Kalman filters may be useful. In other cases, Markov chains may be used. Fortunately, as better unsupervised learning techniques become available, more effective methods for detecting these sorts of anomalies are emerging.

[5] Data diversification has a very long history; Charles Babbage described it and the associated pattern of code diversity in 1837: "When the formula is very complicated, it may be algebraically arranged for computation in two or more distinct ways, and two or more sets of cards may be made. If the same constants are now employed with each set, and if under these circumstances the results agree, we may be quite secure in the accuracy of them." Ammann and Knight offer a more contemporary description in Ammann, Paul E. and John C Knight, "Data diversity: An approach to software fault tolerance", *IEEE Transactions on Computers*, 37(4):418–425, 1988.

[6] See Bianca Schroeder, Eduardo Pinheiro and Wolf-Dietrich Weber, "DRAM Errors in the Wild: A Large-Scale Field Study", Seattle: SIGMETRICS/Performance '09, June 15-19, 2009.

Unsupervised learning

In supervised learning, a set of training vectors is made available to the program being taught. With unsupervised learning models, no training set is provided. In reinforcement learning, for instance, the program receives no help finding patterns beyond punishment for mistakes and rewards for correct answers.

Progress monitors

Progress-monitor programs watch indicators in the system and take appropriate action should progress stop. They often include monitoring for process failures—the most extreme form of non-progress.

Validation techniques

Auditors and regulatory bodies require assurances that isolation boundaries are not breached. Many techniques and tools exist to help demonstrate that these requirements are being met. Bonakdarpour and Fischmeister, for instance,[7] describe the characteristics of tools that, with operating system support, can trace interactions to demonstrate that inappropriate interactions are not occurring in the states invoked by the testing. Common techniques include discrete event simulation, formal design proving, static code analysis and flow tagging.

Discrete event simulation

Discrete event simulation can provide a statistical level of confidence of correctness in cases where it is impossible to provide a proof of correctness of an algorithm, protocol, or isolation—when, for instance, an event distribution is only available empirically.[8]

Formal design proving

If a design can be proven correct and code generated automatically from the proven design, then verification and testing times can be reduced substantially.

The widespread adoption of formal design methods has been hampered by a lack of automatic theorem provers as well as a lack of designers able to handle the mathematical formalism of these methods. Recently, however, tools that hide much of the mathematics have become available, rendering formal design more accessible and practical.[9]

Static code analysis

A weakness of static code analysis is its propensity to generate false positives. Further, the languages we tend to use (for instance, C with its use of pointers) can make static analysis less effective. However, more sophisticated static

[7] Borzoo Bonakdarpour and Sebastian Fischmeister, "Runtime Monitoring of Time-sensitive Systems – Tutorial Supplement", *Proceedings of the 2nd International Conference on Runtime Verification (RV)*, San Francisco, 2011.

[8] See Kleinen, *op cit.*

[9] Jean-Raymond Abrial, "Rodin: an open toolset for modelling and reasoning in Event-B", *International Journal on Software Tools for Technology Transfer*, 12(6): 447-466, 2010.

analysis tools are becoming available, with particular advances in techniques for exploiting contracts in the code, and for symbolic execution.[10]

Flow tagging

Flow tagging can be used to demonstrate that, under the conditions tested, no inappropriate interaction occurred between components. Combined with assertion statements containing linear temporal logic invariants, flow tagging can provide the basis for sophisticated analysis of component interaction and support claims of their isolation.[11]

Deadlock and livelock avoidance

Conditions on the behavior of a system are often divided into safety conditions ("the system never does anything wrong") and liveness conditions ("the system always eventually does something right").[12] That said, the four Coffman conditions needed for deadlock[13] were identified as early as 1971, and today we consider both deadlocks and livelocks as forms of non-progress, liveness conditions.

Proving the correctness of liveness assertions is intrinsically difficult because the search space is, in principle, unbounded: what might the system do in the future? Nonetheless, various tools can help detect deadlocks and livelocks at different stages of system development.

Design stage

Tools are available to help detect deadlocks and livelocks, typically by analyzing an abstract model of the system, including both hardware and software. Some tools can generate code from the design, once it has been proven correct. Unfortunately, if the design is implemented manually, developers may introduce faults into the code and compromise the implementation.

Coding stage

Static analysis tools can sometimes detect potential deadlocks, though this is more difficult for languages such as C than for fully defined and strongly and statically typed languages. These tools perform tasks ranging from extracting semantic information from contracts embedded in the code (as comments in languages that do not directly support programming by contract) to symbolic execution—a cross between dynamic testing and static analysis.[14]

[10] Cristian Cadar, Daniel Dunbar and Dawson Engler, "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs", *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, 2008.

[11] A. Oliveira, A. Saif Ur Rehman and S. Fischmeister, "mTags: Augmenting Microkernel Messages with Lightweight Metadata", *ACM Operating Systems Review* 46(2), 2012

[12] Byron Cook *et al.*, "Proving that programs eventually do something good", *SIGPLAN Not.*, 42(1):265–276, Jan. 2007. See also Edward G. Coffman Jr., *et al.*, *System Deadlocks*, ACM Computing Surveys 3 (2): 67–78, 1971

[13] Edward G. Coffman Jr.*et al.*, *System Deadlocks* by ACM Computing Surveys 3 (2): 67–78, 1971.

[14] See Cadar, *op cit.*

System is running

Many systems provide a hardware watchdog that must be "kicked" periodically. However, it is difficult to determine which process(es) should kick the watchdog, because the watchdog remains oblivious to deadlocks from which it isn't expecting a kick. It is often useful, therefore, to define conditions that guarantee that progress is being made (for instance, a message from process A is received and handled by process B, or the value of an integer is changing monotonically), then place a software process to monitor the condition and take action should progress stop.

Conclusion

No single approach can provide isolation between software components in accordance with the requirements of ISO 26262, just as no single approach can provide adequate proof of this isolation. Used together, however, complementary design and validation techniques can provide an adequate level of confidence that isolation has been achieved. Some of these approaches need help from the underlying operating system, but others can be implemented at the application level.