Justifying the use of software of uncertain pedigree (SOUP) in safety related applications

Peter Bishop, Robin Bloomfield and Peter Froome

Adelard

Abstract

This short paper is intended to serve as an introduction to a publicly available research study undertaken by Adelard for the UK Health and Safety Executive [1]. The main focus for this project was “software of uncertain pedigree” (SOUP) used in safety-related applications. It outlines an overall safety justification approach and ways in which the use of SOUP can be incorporated within that approach. The full report is available from the HSE web site.

Introduction

This paper outlines the findings of a publicly available research study undertaken by Adelard for the UK Health and Safety Executive [1]. The main focus for this project was “software of uncertain pedigree” (SOUP) used in safety-related applications. The paper will discuss the potential benefits and problems of using SOUP for safety-related applications, and then describe an overall safety justification approach that includes consideration of SOUP. We then discuss ways of controlling the costs and risks of using SOUP in safety-related applications both in development and subsequent operation. An example application of the approach is given in the Appendix.

What is a SOUP?

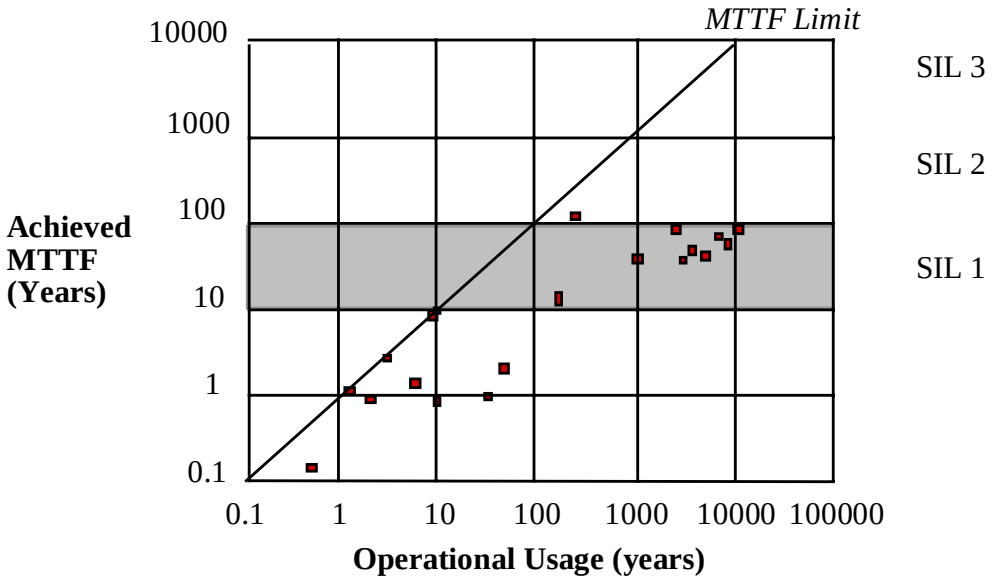
In determining an assurance approach for SOUP, it is useful to be clear about what characterises it. SOUP comes in a variety of forms:

- software components that form part of a program (such as graphic libraries)
- standalone programs and utilities (e.g. compilers and stress analysis packages)
- operating system kernels, networking service, web servers and database engines
- complete systems where the hardware and software are integrated (such as PLCs)

Advantages of using SOUP

Mass market SOUP can reduce the cost of development—indeed, it may be the only way of producing certain systems in a practicable time. Perhaps more significantly from a safety viewpoint, there are good theoretical and empirical reasons for believing that extensive use of a SOUP product will result in increased reliability (as faults are reported by users and corrected by the developers). This is illustrated in the following

figure taken from our earlier research and published in a report to the UK Health and Safety Commission.



Intuitively, one might expect that SOUP could be highly reliable if the following conditions apply:

- small programs
- good quality development process
- extensive field experience
- good fault reporting, diagnosis and correction infrastructure
- stable product

This intuition is supported by a recent reliability growth theory [2], which shows that times-to-failure can increase at least linearly with usage time. The theory predicts that worst case mean time to failure after a usage time t is:

$$MTTF(t) \geq (e \cdot t) / (N \cdot d)$$

where N is the number of residual faults at initial release, d is the number of times the software fails before it is fixed, and e is the exponential constant (2.7181).

Ideally faults should be diagnosed and fixed immediately (i.e. d=1); poor diagnosis (d>>1) has the effect of "scaling up" the failure rate contribution of each fault.

If the software is upgraded with new functions, this introduces an entirely new set of faults ΔN. As these faults will initially have relatively little usage time Δt, the failure rate will be dominated by the new faults, i.e. the MTTF bound for a software upgrade is always less than:

$$e \cdot \Delta t / (\Delta N \cdot d)$$

where Δt is the usage time since the upgrade, and ΔN is the number of new faults introduced by the upgrade. So while reliability improves as "bug fix" versions are introduced, reliability falls at the next major release when new functions are added, and there is no overall growth in reliability for a continuously changing SOUP product (indeed there can be a long-term decline).

The theory supports current engineering intuition, as small programs and a good quality development process reduce N, extensive field experience increases t, good fault reporting, diagnosis and correction infrastructure reduces d, and a stable product avoids “upgrade” effects that limit reliability growth.

Problems in using SOUP

From current theory and available experience, it is possible for SOUP to be reliable but there are also risks in gaining adequate *assurance* that it is reliable, because the SOUP is usually a “black-box” with limited design documentation (although this may be available for “open source” SOUP). In addition the SOUP will often have additional functions that could compromise the integrity of a safety related application. It is therefore necessary to provide a convincing justification for the use of SOUP for a safety-related application.

Safety justification approach

We recommend the use of a documented, phased safety justification that sets out the safety claims for the system, and the evidence and arguments that support them. This is a generic approach that can apply to any system whether or not it contains SOUP.

We identify five safety justification stages that can be required on a real project. They are:

- Preliminary Safety Justification
- Architectural Safety Justification
- Implementation Safety Justification
- Installation Safety Justification
- Operational Safety Justification

These are essentially evolutionary phases in the construction of the overall safety case. The process starts by establishing the claims, then the arguments and evidence are elaborated as the design and implementation of the system progresses.

The characteristics of the safety justification stages, and the main SOUP-specific activities within them, are as follows.

Preliminary Safety Justification - This establishes the system context, whether the safety justification is for a complete system or a component within a system. It also establishes safety requirements and attributes for the system, independently of the technology used for implementation. It defines operational requirements and constraints such as maintenance levels and time to repair.

Architectural Safety Justification - This defines the system or subsystem architecture and makes trade-offs between the design of the system and the options for the safety justification. It defines the assumptions that need to be validated and the evidence that needs to be provided in the component safety justifications. It also defines how the design addresses the preliminary operating and installation aspects for the safety justification (e.g. via maintainability, modifiability, and usability attributes).

The Architectural Safety Justification can be considered at two levels:

- top-level—the assignment of safety requirements to equipment

- computer level—the identification of a hardware/software architecture to meet the safety requirements

Safety properties that might be identified for a system include:

- functional behaviour
- accuracy
- reliability and availability
- fail-safe behaviour
- time response
- throughput (e.g. transactions/sec)
- response to overload
- security (from attack)
- usability (avoidance of human error)
- maintainability (avoid errors when system is modified)

The safety functions and their associated performance attributes will be allocated to programmable electronic systems (PES) and other compenents. Again these requirements are not SOUP-specific, but at the next stage the computer-level architecture will have to address the hazards posed by SOUP within the PES.

The types of evidence available for the safety justification will depend on the computer/software architecture chosen to implement these requirements. The options here could be:

- a complete hardware/SOUP software package (like a PLC) configured by user-level programming (using e.g. PLC logic diagrams)
- off-the-shelf hardware, with various SOUP components, like an operating system, compilers, and library routines
- no SOUP at all (if justification is too difficult)

The choice of implementation approach will be driven by:

- the cost of implementation
- the cost of obtaining evidence
- the adequacy of the safety arguments and evidence for the specified safety requirements (typically more diverse and better-supported evidence is needed for more stringent safety requirements)
- the cost and feasibility of maintaining the arguments and evidence over the system lifetime

Implementation Safety Justification - This safety justification argues that the design intent of the architectural safety justification has been implemented and that the actual design features and the development process provide the evidence that the safety requirements are satisfied. This stage might include results and analyses planned for SOUP components (e.g. to provide additional evidence), but all results would be treated in a broadly similar way.

Installation Safety Justification - This stage needs to demonstrate that the installation is consistent with the design and that operation and maintenance procedures are implemented. In the case of SOUP, this would include appropriate mechanisms for reporting faults, and procedures for dealing with new faults. The process differs from “in-house” software, as there may be no direct method for fixing faults, so “work-arounds” may need to be identified and justified in the operational safety justification.

The Installation Safety Justification also defines any safety-related operational procedures identified in the previous safety justifications. Human factors related issues are addressed such as staffing requirements and competence levels, training of operators and maintenance personnel, and facilities for long-term support.

This safety justification stage also records and resolves any non-compliance with the original safety requirements.

Operational Safety Justification - This reports on whether safety is being achieved in practice. It reports on compliance to operating and maintenance assumptions. It identifies areas where system changes may be required (for technical and safety reasons). It updates the safety justification in the light of changes.

To support this safety justification stage, some mechanism has to be identified for:

- ensuring that the operational and installation constraints are implemented (e.g. by documented conditions of use, training, etc.)
- monitoring the performance of the operational system to identify safety problems for future correction

In the case of SOUP, additional evidence may be obtained from other users of the SOUP (providing there is an adequate fault reporting and dissemination infrastructure) so that latent faults in the software can be identified from a broader range of field experience.

Safety justification of an architecture containing SOUP

The design choices made at the architectural design stage have a major impact on the safety assurance of systems containing SOUP. The choices should be determined by:

- the adequacy of the available safety evidence and arguments
- the cost of obtaining additional safety evidence
- the cost of maintaining the evidence over the system’s lifetime

A “design for assurance” approach within the architectural safety justification can help to minimise costs while maximising safety. For each candidate architecture, a hazard analysis should be carried out to identify the dangerous failures of the architectural components including failures of SOUP. Methods for limiting the effect of such failures should be identified, e.g.

- partitioning (functional or physical isolation of SOUP)
- “wrappers” (interfaces restricting the use of SOUP features)
- diversity (implementation with diverse SOUP)
- safety and credibility checks (checking SOUP results in a wrapper)
- external safety checks and interlocks (to limit consequences of failure)

- dynamic safety checks (e.g. software watchdogs/dynamic test inputs to reveal software lock-ups).

Alternatively, evidence can be produced to demonstrate that the probability of failure is acceptably low so that no run-time defences are required. The cost and safety of the candidate architectures can then be assessed, including the costs of developing and maintaining the safety justification.

Evidence profiles for SOUP

Where a user standardises on specific SOUP, pre-existing evidence can be used in any safety justification that uses that SOUP—this can help to reduce the cost of constructing the overall safety justification. We recommend the compilation of an "evidence profile" for each SOUP component. The evidence profile summarises the available types of safety evidence for a SOUP component, and where available may include:

- test evidence
- analytic evidence (of the product and the implementation process)
- field experience (if the product has been used in former applications)

This profile can include "trusted third party" evidence, where the SOUP product or process has been assessed to some accepted criterion (e.g. TÜV certification, IEC 61508 assessment, ISO 9001-3 process certification, etc.).

The profile can also include information about the stability, usage time and software size. This information can be used to make an estimate of the expected program reliability based on the worst case bound reliability theory [2] discussed earlier.

Safety evidence for SOUP may be black box evidence (e.g. testing and field experience), or white box (e.g. analytic evidence). In many instances it may be possible to obtain adequate evidence by treating a SOUP component as a black box, and the report [1] contains criteria for deciding when black box evidence is sufficient, and when white box evidence is required.

The rigour of evidence produced will also have to be commensurate with the required safety integrity level of the application. Generally speaking the rigour will increase with integrity level, and some forms of evidence may be viewed as essential at the higher levels (e.g. analysis of the source code).

Additional evidence for SOUP

Clearly when there is insufficient pre-existing evidence available for the SOUP (and this is often the case for high integrity applications), it will be necessary to produce additional evidence by testing and analysis. This can be expensive compared with a specially engineered product that was designed to comply with the relevant engineering standards. These additional costs have to be included in the decision to use SOUP components to implement safety-related functions.

Any additional analysis and test requirements for SOUP are fed to the implementation stage. The implementation safety justification has to assess:

- whether the planned tests and analyses have yielded the expected results
- whether evidence is deficient

- the safety impact of any deviations
- whether changes are required to the safety arguments, or additional evidence is required
- whether installation or operational constraints have to be imposed to maintain safety (e.g. interfacing requirements, limitations on mission times, operational procedures, requirements for re-calibration and maintenance)

Some of these issues will have to be re-addressed in the Operational Safety Justification if the SOUP used in the safety application is upgraded over time (e.g. to support new hardware or software functions). Methods for minimising the long term risks of SOUP are discussed in the next section.

Long term risk management of SOUP

Long-term management of safety-related SOUP can be used to limit the risks associated with their use. This includes strategies such as:

- standardisation on a limited number of SOUP components and suppliers
- gaining access to SOUP fault histories for “early warnings” of SOUP problems
- organisation-wide data collection and dissemination of SOUP problems
- a phased strategy for introducing new or updated SOUP within the organisation (i.e. use in low integrity applications first)

Such a controlled approach to the use of SOUP has the advantage that it generates evidence of satisfactory operation, so that there is a stronger “evidence profile” available when constructing a case for the next safety-related application. In addition, the standardisation of SOUP and SOUP configurations implies that one can re-use safety arguments for new systems (e.g. have component safety justifications for items such as operating systems and compilers).

Closing remarks

This paper has outlined a generic approach for justifying the safety of systems implemented with SOUP. An example application of this approach is given in the Appendix. For further details, please refer to the full version of the report [1] which is available on-line from the HSE web site.

References

[1] P G Bishop, R E Bloomfield and P K D Froome, “Justifying the use of software of uncertain pedigree (SOUP) in safety-related applications”, Health and Safety Executive Contract Research Report, CRR 336/2001, ISBN 0 7176 2010 7, HSE, May 2001, http://www.hse.gov.uk/research/crr_pdf/2001/crr01336.pdf

[2] P G Bishop and R E Bloomfield, “A Conservative Theory for Long-Term Reliability Growth Prediction”, IEEE Trans. Reliability, vol. 45, no. 4, Dec. 96, pp 550–560

[3] HMSO, *The Safety of Operational Computer Systems*, HMSO 1998

Appendix: DUST-EXPERT safety case example

To illustrate how this proposed approach to SOUP is applied in practice we will take parts of the DUST-EXPERT™ advisory system safety case as an example. DUST-EXPERT is an expert system that advises on safety measures for handling explosive dusts in industry. The full safety case contains a more detailed justification than that presented here, but the shortened description below shows how the safety case fits into our proposed approach.

Preliminary Safety Case

At this stage we identify the relevant safety attributes for the advisor from the list given in the paper. The selection of safety relevant attributes and the associated safety claims are shown in the table below. Note that not all safety attributes apply, e.g. it is an offline advisor so attributes such as real-time response, throughput and availability have no safety relevance.

	Safety claim
1	Functional correctness (must implement the specified dust explosion calculation)
2	Accuracy (the results are sufficiently accurate when calculated using finite-precision arithmetic, and numerical instability should be detected)
3	Security (appropriate steps are taken to prevent malicious and accidental changes to methods and data)
4	Modifiability (the chance of maintenance-induced errors is minimised)
5	Fail safety (there is a low probability of unrevealed failures)
6	Usability (the system makes it hard for users to make errors)

These claims had to be justified to SIL 2, which implies that the probability of a dangerous result is between 10^{-2} and 10^{-3} per consultation.

Architectural safety case

The chosen architecture contained the following SOUP components:

Component	**Function**
Microsoft Windows	provides windowing and operating system services
IFAD toolbox	used for VDM specification of application (and to generate test data)
LPA Prolog	used to define rules for the expert system kernel
Microsoft C++	used to program the graphical user interface (GUI) for the advisor
Microsoft Visual test	used to automate GUI tests

The main element in the “evidence profiles” for these products was an extensive user base and supplier track record (although for Prolog and C++, fault histories were available). In the following table, the hazards of the SOUP and the associated defences are identified. Note that these include defences in the development process that detect failures in off-line SOUP tools.

Tool	Dangerous failure consequences	Defences
IFAD toolbox	Failure to meet requirements	Acceptance tests, animation of specification, proofs of safety properties
	Failure to provide truthful oracle	Low probability of compensating fault
	Failure to detect type errors	Checks at Prolog level
LPA Prolog system	Faulty code	Diverse checking by static analysis, acceptance tests, statistical tests
Microsoft Visual C++	Faulty code	Diverse checking by static analysis, acceptance tests, statistical tests
	Failure to detect untested C++ code	As above
Microsoft Visual Test	Failure to detect failures during testing	Manual testing on Windows 3.1 version, tests by HSE
Prolog static checking tools	Failure to detect some faults in Prolog code	Acceptance tests, statistical tests
Prolog test coverage harness	Failure to detect untested code	Acceptance tests, statistical tests
Microsoft Windows	Failures to display, perform file access, etc.	Detectable by user as “crash” or freeze.

In addition to this there were a number of defences built into the application design and development process.

Development features	Comment
SIL 2 development process	To aid correctness
VDM specification	To aid correctness of spec, and statistical test data
Statistical testing	Statistical tests to show the 10^{-3} failure target is met.
Directed testing	To ensure that all Prolog code is tested

Design Features	Comment
Feedback of user-specified input	Reveals data corruption in the GUI interface
Interval arithmetic	Reveals unstable calculation method
Databases for explosion data and calculation methods	Permits easy modification for new types of dust, or explosion calculation methods
Access controls	Ensures databases are secure from unauthorised changes

Implementation Safety Case

This provided:

- evidence that the SIL 2 process was followed (documents, audits, etc.)
- results of the directed tests and statistical tests

Installation Safety Case

This marshalled all the safety case elements for the client, and ensured that appropriate installation and operation documentation was available to the users. The overall assurance of safety properties and the amount of diverse evidence used to justify the safety properties is summarised in the table below. Note that the bracketed comments identify cases where the assurance applied to specific parts of the system functionality.

Assurance evidence	**Attribute**					
	functional correctness	**accuracy**	**security**	**modifiability**	**fail safety**	**usability**
directed testing	• (methods & GUI)	• (methods)				
statistical testing	• (methods & GUI)	• (methods)				
analytical arguments	• (VDM & Prolog source)					
desk checks		• (database, methods & warning screens)				
field data	• (run-time system)					
interval arithmetic		• (methods)		• (will detect instability in new methods)	•	
design diversity					•	
prototyping	• (of GUI)					•
stress/ overload testing					•	
manual checks	•	•				
access control			•			
database for methods and explosion data				•		

Operational Safety Case

As part of the ongoing maintenance for the product, the safety case is updated in the light of changes. This includes changes to (or reported faults in) SOUP components that affect the run-time software, e.g. changes of operating systems or C++ versions. The safety case would justify any changes and present the results of the statistical tests to demonstrate that the integrity of the expert system is maintained.