Australian Government
Department of Defence
Defence Science and
Technology Organisation

Reliability Engineering for Service Oriented Architectures

Michael Pilling

Command, Control, Communications and Intelligence Division

Defence Science and Technology Organisation

DSTO–TR–2784

ABSTRACT

This paper reviews the state of the art in Software Reliability Engineering (SRE), and adapts these methods for use in Service Oriented Architecture (SOA). While some prior work has been done on using SRE for SOA, it is incomplete in terms of whole of development life cycle methodology. We outline how existing complete methodologies would translate to SOA and provide a surprisingly simple way of applying these methods to certify the use of legacy software in SOAs. This paper provides a proof of concept but further work needs to be done to elaborate these methodologies for application in SOA.

Published by

DSTO Defence Science and Technology Organisation
PO Box 1500
Edinburgh, South Australia 5111, Australia

Telephone: (08) 7389 5555
Facsimile: (08) 7389 6567

AR No. AR-015-477
February 2013

APPROVED FOR PUBLIC RELEASE

Reliability Engineering for Service Oriented Architectures

Executive Summary

This report looks at how Service Oriented Architecture (SOA) based systems differ from other types of Defence software systems and discusses the important issue of Software Reliability Engineering (SRE) for SOAs.

SRE is needed to be able to procure and predictably deliver Defence software systems based on SOA that can be guaranteed to operate successfully (within specifications). This is necessary to ensure they can be relied on in battle.

The report examines the current state of the art in Software Reliability Engineering and shows how aspects of existing work can be applied in the SOA context.

In particular it shows how to use SRE to certify:

- The core SOA infrastructure,
- Services composed from other services,
- Applications, and
- Importantly, how to certify legacy systems for incorporation into SOA applications, SOA systems or systems of SOA systems.

The report provides several recommendations for enabling SRE for SOA in Defence and also shows how Reliability Certification provides a clear and transparent method for acceptance or rejection of software deliverables by the Australian Defence Force.

Key recommendations of the report are:

- Defence adopt automated testing for SOA based on black box specifications and usage models allowing for automatic test oracles and automated testing of new software components, software compositions and applications. Governance should ensure that such specifications and usage models are recorded for each entity in the software hierarchy. Such techniques have driven a ten fold reduction in the measured incidence of software defects for the US DoD.
- Defence SOA computing nodes should be able to operate in three modes: *test*, *normal* and *war/high reliability*, each having stricter reliability certification requirements than the previous.
- Governance is crucial for a well functioning SOA environment and should include provenance for both software and data. This is essential for quarantining problems and for finding the root causes of problems.
- The software repository for the SOA environment should record for each component and software composition/application its current synthetically tested reliability estimate and field tested reliability measure so that decisions to include certain software in a system can be made in an informed manner.

Author

Michael Pilling
C3ID

(U) Michael Pilling completed a Bachelor of Science degree with honours in 1987 and a Ph.D. in Computer Science in 1996 at the University of Queensland, Australia. Michael's specialities are distributed and real-time systems, job scheduling, formal specification and program correctness, criticality management, and the calculus of time. His current interests include Software Reliability Engineering, Failure as a fundamental construct in usable and effective systems, Virtual Synchrony and its application to synchronous group communication, performance engineering of computer systems, and graceful degradation of systems in the face of failure and overload.

Contents

Figures

THIS PAGE IS INTENTIONALLY BLANK

Glossary

ADF Australian Defence Force

Choreography A method of creating a complex service out of simpler ones in which no party in the interaction is aware of anything but its own part to play, see section 3.

CIOG Chief Information Officer Group

COTS Commercial Off The Shelf software. Commercial software that is pre-written for generic requirements of a particular domain.

CORBA Common Object Request Broker Architecture

Ecosystem In software, an ecosystem is a set of applications and/or services that gradually build up over time to provide a rich and diverse offering of functionality. An SOA's economic value is derived from the utility of the ecosystem that grows in it.

ESB Enterprise Service Bus

Foreign In an SOA context: Any SOA, service or software which the owners of the calling software do not have control of, either in terms of infrastructure, code and debugging, or operation.

GOTS Government Off The Shelf software. Like COTS but built and owned by Government.

GUI Graphical User Interface

Open Source Software written by others for which the source code is published and thus able to be modified and adapted by the end user under prescribed legal conditions.

Operational Profile A detailed usage context for software being considered consisting of a complete set of significant operations along with their probability of occurrence.

Orchestration A method of creating a complex service out of simpler ones using a central controller, see section 3.

NGO Non-Government Organisation, usually charities or non-profits but also businesses.

QoS Quality of Service

SOA Service Oriented Architecture

SRE Software Reliability Engineering

System Mode Many systems exhibit different modes of operation. E.g. the cockpit automation of a commercial airliner will have different active tasks during each of its modes such as Taxiing, Take-Off, Landing, Climb and Cruise. Defence systems effectively have peace-time and war-time modes of operation although these may be factors of usage rather than consisting of different sets of programs.

Usage Model A usage profile or operational profile plus a statistical model of how likely each significant operation is to follow any other significant operation.

Usage Profile See operational profile.

W3C World Wide Web Consortium

WSDL Web Services Description Language

XML Extensible Markup Language

1 Introduction

1.1 CIOG Mandate

CIOG has mandated that future Defence information systems will be built using an SOA infrastructure[CIO10, CoA11].

A key driver for SOA adoption is the expectation of being able to compose new services from existing lower level services, not only allowing reuse of software and so lower software build costs but also enabling access to a wide variety of Defence and whole of government data for uses that were not envisioned when the original systems were constructed.

However, deployed and peacetime Defence Systems such as those monitoring, assessing and ensuring the preparedness of the ADF for potential deployments must be stable and reliable. Given that SOAs will be the core platform of the future Defence Information Environment, this paper looks at how Defence can determine and certify the reliability of SOA-based systems so they can be deployed with confidence in their stability and correctness.

1.2 What is an SOA and why is it different?

When CIOG issued a Defence instruction stating that SOA is now the preferred architectural style for the Defence information environment, it defined SOA this way:

> Service Oriented Architecture (SOA) represents an architectural style that aims to enhance the agility and cost-effectiveness of delivering IT capability within an enterprise while simultaneously reducing the overall risk and maximising the organisational investment in its IT capability. It accomplishes this by encapsulating technical capability as one or more business services that are used and re-used throughout the enterprise. SOA supports service-orientation through the realisation of the strategic goals represented by service-oriented computing. For example, some key SOA goals include risk reduction, agility, and existing technology investments.[CoA11]

There are many definitions of what constitutes a Service Oriented Architecture (SOA), however the following one:

> SOA is an architectural paradigm for dealing with business processes distributed over a large landscape of existing and new heterogeneous systems that are under the control of different owners.[Jos07],

is highly relevant for the Australian Defence context because of the need to integrate many legacy systems, to interoperate with other forces and increasingly to interoperate with other Australian Departments, both State and Federal, as well as with businesses and NGOs for counter-terrorism operations and civil emergencies.

Other relevant definitions emphasise platform independence and loosely coupled interoperability[SOA06]; visibility of interactions and their effects[Oas06] and implementation diversity and whole of life-cycle management[NL05].

Many people appear to equate SOAs with Web Services[1] but Web Services, in and of themselves, are neither necessary nor sufficient to create an SOA. Other distributed technologies such as message queues can be used to build an SOA and it is the organisation of interfaces around self-contained business process steps that most strongly distinguish SOAs from other distributed architectures which are usually implementation centric[2]. It is these relatively coarse-grained independent business process steps that lend themselves to composition into many different business applications. SOAs are focused on creating end user business value.

Nor do distributed objects such as those commonly provided by CORBA, of themselves, constitute an SOA[Jos07]. Experience shows that such remotely accessed objects are code-centric rather than business process-centric and generally result in complicated, highly coupled applications with lots of dependencies that will not scale.

The focus on business processes and value affects the applications and structure of SOAs at every level. In particular, as individual SOAs age and develop, they almost inherently end up being heterogeneous implementations as more of the organisation's existing infrastructure is linked into the SOA. While the organisation may seek to constrain this heterogeneity, and this is a useful goal in terms of reducing complexity to manageable levels, eventually a merger or partnership interaction will force the SOA to accommodate foreign technologies. Likewise any SOA that is running in real-world deployment will eventually end up with multiple versions of the same service co-existing as upgrades occur.

To meet the challenge of providing a usable and reliable yet diverse operating platform for Defence, system specialists will have to integrate components, including legacy components, into working applications that can provide semantic, operational and timing guarantees to their users or other higher order components.

SOA systems are also inherently dynamic. Not only are their components loosely integrated and bound together by run time scripts but also these components may be in a state of flux as they are modified, upgraded, replaced and retired.

Clearly Defence Systems built on SOA will need to be reliable, but SOA throws up some particular challenges for building and certifying reliable systems. These include:

- The integrated grid of Defence SOA platforms may be sprawling and also highly diverse, how can one talk about reliability for such a system?
- How can one be sure a Defence System built on SOA is reliable enough to deploy?
- Some components of Defence SOAs will be wrapped legacy systems, how can and should these be certified?

[1] Processes called by each other over the internet or similar infrastructure in a call-response manner using protocols such as Http or SOAP.

[2] I.e. they have interfaces that strongly embody the implementation, dealing with information technology concepts rather than business concepts.

- How can one deal with components built and / or maintained by others and still get a reliable system?
- What can be said about the reliability of services created by composing other services?
- How do multiple versions of the same component affect reliability and the calculation and certification of reliability?
- What can one say about the reliability of compositions that include foreign services, i.e. services provided by other organisations or sub-organisations of Defence itself? (The technical issue here is that the part of Defence operating **this** SOA instance has no control, only influence, over the "foreign" service. While foreign services may be provided by subcontractors, other government departments or even enclaves of Defence itself; a special case of foreign services for Defence are those services provided by its coalition partners in a war, an exercise, or for peace-time sharing of information and services.)

1.3 What do we mean by Reliability?

Musa, Iannino and Okumoto produced what is now a standard definition for software reliability.

> Software Reliability is the probability that a system or a capability of a system will continue to function without failure for a specified period in a specified environment. "Failure" means the program in its functioning has not met user requirement in some way.[MIO87]

This is a quite specific definition that still allows considerable latitude so that software reliability can be expressed in terms meaningful to the system users. For instance, the period can be a length of time that is natural to the system: days for a call service centre or minutes for a missile guidance system. While system execution time is the fundamental denominator of software reliability, using proxies for system execution time allows software reliability to be easy to calculate and to be represented in a scale that is meaningful to humans and compatible with measures of hardware reliability. Such proxies for system execution time can be the wall clock time as above or the number of invocations of the system or the number of executions of a core system function such as database lookups in a retrieval system. So measures may be a 1% chance of failure per hour, or a 0.5% chance of failure per 100,000 database accesses. A *failure* is as the user would define it, so anything that prevents the system from being workable is definitely a failure; however it is arguable whether a message being presented in the wrong colour is - it would be if it were a safety critical warning message and the colour failed to draw attention to the message or contradicted a standard safety colour requirement. While the above definition may sound loose at first glance, it is tied down once it is instantiated for any particular system. In general things that are arguable failures in a particular context are something that users in that context can live with, whereas true failures are things that users cannot tolerate. Of relevance to Defence, Musa notes that software safety is a specialised subcategory of

reliability[Mus04]. The above definition of software reliability was deliberately designed to produce a measure that is numerically comparable to hardware reliability enabling a single reliability for combined hardware and software systems to be calculated.

Note that reliability is always greater than availability, as availability is reduced by the time to diagnose, repair and/or reboot after a software or hardware failure.

1.4 Other approaches to Reliability

We specifically don't consider important approaches to creating higher reliability such as software redundancy, redundant software output voting, hot fail-over or other fault tolerance techniques, firewalling and circuit breakers[Nyg07]. These techniques enable us to build systems with higher reliability out of lower reliability components, but they rely on those underlying components still having some minimum and known standards of reliability in order to be configured correctly and generally cannot rescue a deeply unreliable system. These techniques are predicated on a slightly different concept of reliability in which the underlying components do fail but that failure is recovered from so the macro component is seen as reliable. They also do not give any insight into how to measure the reliability of the underlying components or how to drive increases in their reliability. Equally, we do not cover cases where the native reliability is degraded by security attack, or is undermined by the SOA making bad data more available[PSRF09]. We prefer to focus this paper on the necessary problem of calculating and improving the reliability of the core software components and any products made from them using non-redundant calling. Successfully solving this problem will allow these other techniques to be applied in ways that allow the augmented reliability of the redundant service to be calculated given the component reliabilities.

1.5 Data correctness is an important issue this paper does not address

Even with correctly functioning programs, an SOA can deliver incorrect results if the data it operates on is incorrect or inappropriate to the problem[TWZ+07, Fis09]. We regard these as separate problems, namely of correctness of specification, intent and quality of data capture and encoding; rather than the problem of component correctness and system reliability quantification we are investigating.

1.6 Uncontrolled (Foreign) Domains

We are particularly interested in how to analyse systems that by necessity make calls to services resident in SOA domains that Defence itself does not control. A methodology for building and certifying Reliable SOAs cannot be complete without being able to handle this problem.

1.7 Composite Applications

Likewise a primary, if not defining, feature of SOAs is composite applications. These are applications built by temporarily combining existing components through orchestration or choreography to produce a bigger application. The differences between orchestration and choreography are dealt with in detail in section 3. A composition may be performed manually by a human writing an orchestration or choreography script. More rarely, as the technology is still underdeveloped, they may be automatically composed by the system.

This brings us to the static-dynamic composition spectrum. Component services may be listed in a white pages service, where they can be looked up by name; in a yellow pages service, where they can be looked up by function; and in a green pages service where their interfaces and calling protocols can be looked up. Even in the white pages, a specific name may lead to several versions of the service ranging from obsolete to current to experimental / in testing. Depending on whether particular services are hardwired into human written scripts or whether they are looked up in such services, and whether a specified instance of a replicated set of services is called; the same script may invoke the same service instances every time or it may invoke completely different service instances each time or even instances derived from different code bases.

Our methodologies for quantifying reliability must take these possibilities into account.

1.8 What this paper seeks to achieve

"Service modelling and service-oriented engineering — service-oriented analysis, design and development techniques, and methodologies — are crucial elements for creating meaningful services and business process specifications[PTDL07]."

We agree and believe part of these engineering techniques are methodologies for meeting reliability requirements in SOA systems. This paper seeks to show:

- How SOA reliability might be engineered in,
- How SOA component reliabilities might be measured
- How reliability can be maintained or certain reliability levels reached when some components are a given or even foreign[3].

We wish to be able to calculate reliabilities required for components, measure their actual reliability during system building and validate or certify that those reliabilities have, in fact, been achieved.

2 Review of Software Reliability Engineering (SRE) Approaches

In this section we examine existing approaches to SRE to inform our development of tailored approaches to SOA SRE. Before reviewing current Software Reliability Engineering

[3]any code that is not written by this organisation including COTS, GOTS, Open Source and Freeware

approaches, it is important to realise that it follows from the definition of reliability that every assessment of reliability must be against an explicit or implicit specification of the software's correct behaviour. We also need to distinguish between faults and reliability. Each fault in the software may or may not be severe enough to cause a failure. For instance the earlier example of a message being printed in an unexpected colour may never qualify as a failure in some instances, whereas a memory leak fault may need to be activated thousands of times before it accumulates enough damage in the running system to cause a failure. Faults must be activated to become apparent. For this reason reliability is a function of the *actual* usage of the system, and so testing and certification must be performed with respect to an expected usage of the system. It is likely that over time the actual usage will diverge from original expectations. This is especially true of core infrastructure in an SOA. If the usage of the system never activates a fault, it cannot be the cause of a failure and so the fault's existence is irrelevant to the reliability of the system in that usage context. For this reason the reliability of a software product can change when it is re-purposed, such as when a component is reused in a new service composition.

Since it is simplistic to equate faults with failures, it is incorrect and misleading to consider measures of fault removal to be indicative of the reliability of a system.

2.1 Statistical Analysis based on Markov Chains

One significant approach in reliability engineering is analysis of software systems based on Markov chains. A Markov chain is a representation of a system that has multiple potential states using a transition diagram in which nodes represent states and edges represent transitions between them. A defining feature of a Markov chain is that the choice of transition out of any state is a function purely of its current state and an input, not the history of previous states. Some transitions may return a Markov chain to a previously visited state. This approach arose from Cleanroom Software Engineering[PTLP99] which requires that all software be rigorously specified. The usual form of Cleanroom specification is to describe a series of user observable software states and what stimuli causes the transitions between them. A good example is how pressing the decoration buttons on a GUI window affects the display of a window, or what delivery or loss/time out of a communications packet does to the state of a sliding window communications protocol. The important thing is that each state in the Markov chain does not record the entire state or history of the system, just the unique state among some subset of attributes[4]. In the case of this type of reliability analysis, these attributes will most often be user observable and will at least be test system observable.

Figure 1 shows a usage Markov chain for a usage most readers are familiar with - that of using a bank ATM, and also presents a good example of how transition choices that aren't truly independent of history, can be made to appear so and not affect the validity of analysis outcomes. Although the success or failure of a withdrawal is dependent not only on the current amount requested but also on the history of deposits and withdrawals, the bank can provide accurate statistical measures of how many withdrawals fail due to

[4]Researchers are aware that sometimes system state does depend on the entire history of the system, however most histories map well onto a single state and so it is not an unreasonable model assumption. Software Reliability Engineering experience also shows this model simplification has worked well.

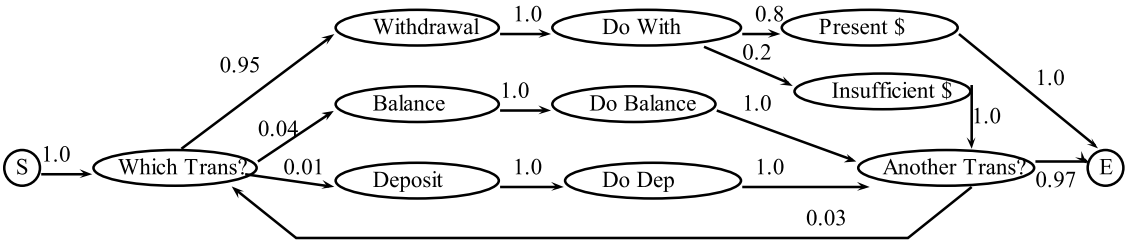

Figure 1: *A usage Markov chain showing transition probabilities for an auto-teller*

insufficient funds so we can build an accurate usage chain and preserve the assumption of no history. This figure also illustrates how a fault can affect reliability differently depending on its location. It's clear from even a cursory examination of this usage Markov chain that a fault in the *Withdrawal* code that incorrectly rejects a valid withdrawal is going to cause an order of magnitude more failures than a fault in the *Deposit* code that prevents valid deposits. Mathematical analysis of the model can produce exact failure comparisons for these two fault locations.

Whittaker and Poore[WP92] describe a process for reliability analysis comprising of the following steps:

1. A *usage Markov chain* is constructed from the system or subsystem specification. The state and transition graph is first constructed from the specification, then each transition from each node is assigned a probability so that the total probability of leaving each node except the terminal node is 1. Transition probabilities can be assigned in multiple ways:

   - In the absence of knowledge all transitions leaving a node are assigned equal probability. This is the *uninformed* approach.
   - If some record of usage sequences is available, these are used to derive the relative probabilities for each transition This *informed* approach is most common when a prototype or prior system exists.
   - In the *intended* approach probabilities are assigned by hypothesised behaviour of a reasonable user or expected system driver.

   In this way, the usage Markov chain embodies the expected usage of the system to the extent it is known. Such usage chains can be built to the level of granularity appropriate for the problem, usually they would reflect transitions between significant modules not individual control structures in programs.

2. The usage Markov chain is used to drive testing by generating random inputs that conform to the expected usage distribution of the system.

3. A *testing Markov chain* is constructed by initially copying the nodes and transitions of the usage Markov chain, and each transition is assigned an initial count of zero.

4. During each test when a transition is traversed, its count is incremented in the testing Markov chain.

5. If a failure occurs in state ‘X’, a transition is created from ‘X’ to a new state ‘fail’ with a count of 1. Further failures add new transitions to ‘fail’ or increment the transitions count if it already exists. In this way the number of failures in each state can be quantified over a series of test runs. When a failure occurs a decision must be made whether to continue testing or fix the bug. If the failure is not serious enough to have to stop the test, a transition is then made from ‘fail’ to the next state it recovers to.

6. Mathematical formulas are given to analyse the counts in the testing Markov chain to derive values for reliability and mean-time to failure.

Poore et al.[PMM93] adapt this approach to component based software engineering. In their Markov chains, each node represents a component and transitions represent passing control from one component to the next. They make a clear distinction between planning a software system build, and certifying it to a particular reliability level. In the former case, estimated reliabilities of components are used to adopt or reject them and the estimates can be quite crude. Essentially one must choose the best alternative, including the alternative of a new build of that component. In the latter case, the reliability estimate has to be accurate enough to decide to field a system that may be mission critical. Their system is based on four basic ideas:

- Systems are composed of components.
- Component reliability can be measured.
- System reliability can be calculated from component information.
- System certification may be based on a different model from that used in reliability planning.

Component reliability for component selection can be derived from the past history of that component. If it is a new component, its reliability might be estimated from evidence such as the past performance of the programmers that produced it, or the quality of the software development techniques used, or evidence from a testing regime.

The Markov chain model described is referred to in the paper as the component model and can be used to perform what-if analyses of the effect of modifying the reliability of particular components. This allows the system designer to see that the reliability of the system may be very sensitive to the reliability of certain components and comparatively insensitive to that of others. This can guide software development efforts. One can also set a system reliability and derive allocated reliabilities for each component needed to meet that target. The transition probabilities for the component model must be estimated on the basis of design and intended use, and so humans must decide on the likelihood of each usage.

Poore et al. give two other models and associated mathematical formulas for determining reliability.

- A *sampling* model for estimating the reliability of existing components or subsystems without regard to their internal construction. It involves generating system inputs

from a distribution of expected usages and testing the component until a certain confidence level in the target reliability is reached or the component fails to meet the reliability.

- A *certification* model which accumulates data across multiple versions of components to reach a certifiable reliability rapidly. It makes the statistical assumption that the mean time to failure grows exponentially over successive versions of the system and requires that components that are corrected for bugs are regression tested before reintroduction to the system for further tests.

Some advantages of the Poore et al. approach are:

- Some guidance about which components will be critical to a system's reliability will be discernible even without accurate information about the usage of the system. These can be used to guide software development until a prototype is deployed and actual usage data collected.
- The derivation of reliability from the model does not make assumptions about the distributions of faults within the software.

Many aspects of Cleanroom Software Engineering make it attractive as a methodology for developing reliable SOA based systems. It has a very strong mathematical and statistical basis that not only monitors the development of the software, it also monitors the development of the production process. A key approach of Cleanroom is to start with high reliability and stay high. It does this by starting with very small core parts of the system and get them fully working before adding the remaining functionality incrementally. The motivation for this is to keep developers enthusiasm up, as well as be able to accurately state how finished the system is: "At the end of the first increment, for example, one can be confident that 20% of the system is 100% complete, rather than speculating that 100% of the system is 20% complete[PTLP99]."

The discrete component nature of SOA makes incremental development a natural fit and there are many benefits to a having a working system even if it is pared down. Other components can be developed against the completed parts of the system and because the completed parts of the system have been thoroughly tested, people doing the development can be reasonably confident that they will be debugging their own code rather than other's.

The usage Markov chains not only provide a way to guide development to focus attention on the components that need to be most reliable, they allow for the automatic generation of test cases that will match the usage of the system.

2.2 Musa's Approach

In his textbook[Mus04], the late John Musa details a complete development process for Software Reliability Engineering (SRE)[5] of systems in general, before the advent of service oriented architectures. His approach is as follows:

[5]Software Reliability Engineering is both the name of Musa's methodology and a generic term much like "Hoover".

- Define the *product*, including who are the suppliers, customers and users. Identify the base product and any *variations*, such as the same system provided in a different language, as well as associated systems that must be tested separately. A variation may be due to: a substantially different set of users (so the variations will have different operational profiles); a different operating environment; a different software implementation[6], a different hardware operating platform, a different change approval process (so the actual operating set of components will skew over time).

  A *supersystem* is "a set of interactionally-complex systems that include the base product or a variation plus independent systems, where users judge the product by the set's reliability or availability.[7]" Supersystems may produce very different user apparent reliabilities out of the same base system.

- Develop an *operational profile* which is a detailed usage context for the base product and each of its variations. Generate a complete set of operations along with their probability of occurrence. Each *operation* is a self contained major logical task in the system that is performed for some initiator and returns control to the system on completion. Examples of operations might be "process fax call" in a PABX or "display an employee's leave balance" in a payroll system. In an SOA system, operations would correspond naturally but not exclusively to top level service invocations. Operation occurrence rates can be estimated or preferably gleaned from existing systems, they are then translated into occurrence probabilities. An important step in capturing all operations is to correctly identify the group of operation initiators in the system.

  Operational profiles along with criticality information can be used to direct software development effort in many ways.

  1. Apply the Reduced Operation Software concept[8]. In some cases it is more cost efficient to eliminate low frequency or low criticality operations and use a manual process. In particular, simply reducing the number of operations in the software will increase its reliability as there is less code that could contain faults and the operations remaining will be of higher frequency and therefore better exercised in test and deployment. This may result in a revised operational profile.
  2. High probability operations are prime candidates for reuse and in an SOA or software service system should be exposed as a service if they are not already and they are sensibly reusable business steps.
  3. Use operational development to optimise time to deployment, using the Pareto principle you might implement the 5% of software operations that are used 80% of the time or are the most critical in the first release and so on. This not only allows quick deployment for core functionality, but also ensures that in later releases, the most important operations are already field tested. Musa notes that installation is a critical but incredibly low frequency operation that must be included in the first release.

[6] note e.g. with different bindings

[7] In Defence terms PMKeys might be a base product and PMKeys accessed by Internet Explorer or by Mozilla Firefox would be two supersystems.

[8] The software analogue of RISC hardware.

4. Allocation of development effort should also be proportional to the operational profile including system engineering, architecture, requirements and design effort, coding and unit testing.

Operational profiles are also highly useful when doing Software Performance Engineering [Smi90, SW01, SW04] so the effort in creating them can contribute to multiple outcomes.

- *Engineer "Just Right" reliability.* That is the minimum truly acceptable reliability at least cost, but in a degraded development environment given an insufficient budget to reach that reliability, it may mean accepting the lesser goal of the maximum reliability possible given resource constraints.

  Failure intensity is the inverse of reliability and may be expressed in terms such as 5 failures per 100 print jobs. It is more convenient to work with failure intensity calculations at this point. The major steps in calculating just right reliability are

  1. Define what is meant by "failure". This must be done from a user perspective, which may be influenced by legal or industry standards. Note that it is usual but not necessary that definition of failure is similar among associated systems or components. Failure severity classes may be defined at this point for later direction of software correction efforts, but is not necessary for SRE.
  2. Choose a common reference unit for all failure intensities. Other reference units will have to be converted to the common one.
  3. Set a system failure objective for the main system and each variation. For the overall system, the user may be more concerned with reliability or availability. The former is usually the case when there are dire consequences for it not working immediately - e.g. a fire control system, whereas the latter is usually the case where the operations are slightly deferrable, but throughput or access is important such as a pay system where a certain number of payments must be made per fortnight and for most of the fortnight the deadline is not near. If the user prefers availability, simple calculations can derive the required overall reliability given the time to execute a given recovery procedure.
  4. For any software developed in house or under our control:
     (a) Find the developed software failure intensity objective. Musa calculates this by subtracting the failure intensity of acquired software from the system failure intensity objective. This implicitly assumes that equal time is spent in all software components.
     (b) Choose software development strategies to optimally meet that objective, that is components with stronger objectives would be allocated more time for say design and test, and may also use more rigorous but expensive development methods like formal methods.

- *Prepare for testing.* This involves planning both test cases and test procedures. The key here is to use the operational profile to guide the proportion of effort, and the proportion of actual tests allocated to each operation. Testing involves feature testing which corresponds well with operations, load or stress testing which should be done so the number of simultaneous instances of each operation is in proportion

to its operational profile, and regression testing in which the frequency of retesting operations should be based on their occurrence probability in the operation profile.

It is not envisioned by Musa that the entire regression test suite is exercised on each update, but rather that this is done over progressive cycles of regression tests as each change is made.

Musa advocates manually preparing new test cases or at least the classes of test with automated instantiation of them, rather than simply replaying inputs recorded from field data, so important but rare inputs are not missed. He also warns that automated input playback does not allow for optimisations of test cases which are essentially equivalent and implicitly warns that such playback may mis-sample the operational profile due to a small or local recorded sample. The number of new test cases needed is estimated using industry standard models of test cases needed per 1000 lines of code. Except in the initial version, new test cases relate only to new functionality added and test cases should be allocated according to each new operation's proportion of total new operations, and its operational profile. New test cases are allocated between a base product and its variations on the basis of product market-share among the variants, however if the variants differ only in implementation and not operations then no distinction needs to be made.

- In *executing the tests*, Musa distinguishes between two types of test which are differentiated not by the stage of the testing in which they occur, but by their objective.

  Reliability Growth Test The objective is to find and remove faults. Models are used to estimate and track failure intensity, which in turn is used to guide development and when to release. Reliability growth tests are usually used to system test software you are testing yourself, but it can be used for beta testing if you are getting the failures resolved. This type of testing includes feature, load and regression tests. Load testing includes testing for concurrency errors such as deadlock, livelock and race conditions. Regression testing is used to test whether previously functional parts of the system have been broken due to updates. The only advice on the amount of testing time needed Musa gives is to estimate using the test time of previously tested similar systems.

  Certification Test No debugging occurs in certification testing as it requires a stable system without fixes or added features. The objective is to decide whether to accept the system or return it for rework. Certification does not rely on finding a particular sample size of failures, but on a sufficiently high sample size of operational profile directed test cases. In fact if no failures at all occur for a sufficiently long test regime the software will be accepted. Likewise it is possible to certify a system that does demonstrate failures, provided they are infrequent enough to meet the reliability standard.

  Test time includes time taken to set up, record results, clean up and identify failures. It does not include fault resolution but can be greater than available calendar time if multiple test systems are available. Although the failure intensity reduction we are looking for is calculable, Musa offers no way to calculate the amount of test time needed to achieve that beyond past experience. The available test time is allocated among supersystems based on their fielded market share among themselves, and an

estimate of relative risk between 0 and 1 according to previous results with that system. An as yet untested system has a relative risk of 1.

SRE testing starts after the components have been tested and integrated, and the system can execute as a whole. The testing system can often detect deviations from the specification but it is usually up to a human to determine if a particular deviation constitutes a failure. Failures need to be recorded along with the sequence in which they occur, so that failure intensity can be calculated.

- *Guiding tests.* In this activity data collected from testing is used to track the reliability growth, later it is used to certify the system reliability.

  Tracking reliability growth is done by estimating the ratio of observed failure intensity (FI) to the system or variant failure intensity objective (FIO), this is done using one of several available software reliability estimation programs of which Musa recommends a free example CASRE[Nik02]. Within CASRE Musa generally recommends using both the logarithmic model which is slightly too pessimistic and exponential model which is slightly too optimistic. Together they bound the estimate quite well. Such estimates are made with increasing frequency as available testing time is consumed. In most systems the FI / FIO ratio should drop sharply as testing commences and they slowly trend downwards, this is due to the use of testing driven by the operational profile. Uniform testing generally results in a linear drop in this ratio. Figure 2 shows this graphically, the fault intensity is being reduced as defects are found and software corrections made - spikes in the FI / FIO ratio with SRE occur when new features or bad fixes are introduced.

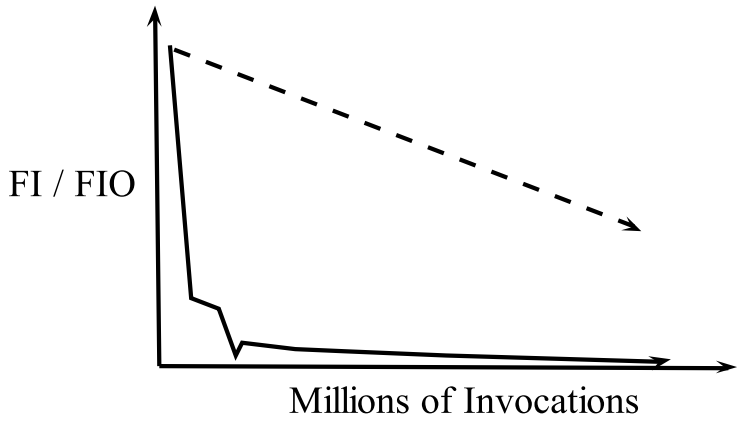

Figure 2: *SRE fault intensity reduction (solid line) vs. with standard testing (dashed)*

Musa recommends that a system is ready for release when its FI / FIO ratio is between 0.5 and 1.5, the range being due to estimation uncertainty in the CASRE models. Other software reliability estimation programs allow the user to set confidence limits for the actual failure intensity estimation and generally Musa recommends the desired failure intensity be tested at 90% confidence.

For certifying reliability, decision charts based on sequential sampling theory are available on which to plot successive failures on their axes: failure number and normalised unit time of failure. By choosing the appropriate chart one can select

the discrimination factor or factor of error in estimating the failure intensity you are willing to accept, the *consumer risk* level or probability that you will accept an unacceptable system, and the *supplier risk* level or probability that you will reject an actually acceptable system. Each chart contains accept, reject and continue testing zones. As each test failure is plotted, the zone it falls in tells the tester what to do with the software. While it is theoretically possible that test failures could continue to fall in the continue testing zone; in practice they cross into the accept or reject zones and the testing is very efficient in that such decisions are made as soon as the minimum number of failures or failure free tests to be able to decide has been reached. For certifying reliability only operational profile driven load testing data should be used. Musa[Mus04] provides the most commonly used charts as well as general formulas for constructing decision charts for any combination of discrimination factor, consumer risk and supplier risk.

Musa's approach has many parallels with the Cleanroom approach but is more heuristic in some areas. He does not focus on software specification at all but is very strong on identifying systems, system operations and their relative usage probabilities. Lacking rigorous specification he has a more user centred concept of failure. One of the contributions of his work highly relevant to SOA is his concept of system variations and supersystems. Whenever a software service is recomposed into a new service composition, it is effectively like a new supersystem for that service. While this may be stretching the idea a little, the point is that users may judge the component based on the performance of the new composition and more importantly that the component will be exposed to a different operational profile originating from the new service composition. It may be exercised in entirely new ways and fail.

His methods for deriving target component reliability are much simpler than in Cleanroom, as he uses simple subtraction. This may well be sufficient for a component before it goes into testing.

Musa's advocacy of Reduced Operation Software Components is a good rule of thumb for service development. While an SOA service should be complete in a business sense, it will be more reliable the simpler the interface and the fewer operations it embodies.

Like Cleanroom, Musa's SRE strongly separates reliability growth from certification testing. There are good reasons to use a cheaper and faster testing regime to guide reliability growth during development and then only expensively test for certification when it is likely to succeed. Moreover, a lack of reliability growth as evidenced by testing can be indicative of problems with the software engineering approach being taken and so is a good checkpoint.

One of Musa's strongest contributions is his insistence on the primacy of the operational profile as a means of directing testing in a highly efficient manner. In SOA, this should serve as a warning that composing a new service out of previously reliable components may result in a lower composed service reliability than expected because the components have been re-purposed.

2.3 Service Oriented Approaches

There has been some work on SRE specifically for SOA but these have been far more divergent than Musa's SRE and Cleanroom which are both complete in themselves and follow similar paths to certification that are different in their detailed processes.

Tsai et al[TZC+04] assert without explanation that the move to web services as opposed to product based software development invalidates many reliability assessment techniques, including ones that assume reliability growth. They provide a novel means of determining component reliability, basically exercising groups of competing components through a voting engine that gives extra weight to components with higher proven reliability, finding fault with components that don't conform. Formulas are given for calculating the progressive reliability of a component as reliability updates bubble through the testing system. They argue that this can allow testing of new components inside operational systems at minimal extra cost. This technique certainly has the advantages of:

- Providing an automated test oracle to validate new services.
- Testing new services under normal operating conditions
- Providing fast or even immediate feedback about component reliabilities as they are further tested.

Some disadvantages include:

- It cannot evaluate the first service of a particular type.
- Reliability of one aspect of a component does not necessarily predict the validity of responses given by that component for another type of query or invocation. Without the calls made to the voting system being aligned with an operational profile, this may result in a misleading sample and hence a misleading result. It is particularly dangerous if an early burst of simple requests are deemed correct and so boost the voting weight given to the new component prematurely.
- It requires the Web Services Description Language (WSDL) signatures and calling sequence of competing services to be identical which may not be the case. The real difficulty would be when a new service has a calling sequence that differs from similar services. While wrappers may be able to include new services with extra or different parameters, it is much harder to wrap a service to change its pattern of invocation to include it in this testing.
- The authors themselves acknowledge that the approach "would not work well when no alternative services are available". The previous point can impact on such alternative service availability.

The assumption that multiple implementations of equivalent services exist is not a given in the Defence domain. While some types of services like geospatial data are provided by multiple suppliers, there can be significant differences between the accuracy or resolution of the data delivered, and the WSDL interfaces and calling grammars of competing services

could be quite different. Likewise, many Defence services are too complicated or monolithic to duplicate with a different code-base - such as the PMKeys pay system or the output of Australia's military and civil radars.

To be able to use this technique in coalition operations, coalition partners would have to agree identical WSDL specifications and calling sequence grammars for services. As described above wrapping such services would only work in cases of simple parameter divergence.

Defence and civilian government data could be accessed at lower levels of aggregation and processing, but it is not clear where alternative services to do this currently exist. Right now this group voting technique does offer great promise in testing new versions or updates of an existing service for incorporation into the system.

Tsai et al. also provide formulas for calculating the combined reliability of a composition given the composition of its components. In particular, they provide a formula for calculating the reliability of a replicated set of components which each return the same result[TZC+04].

In his 2005 paper, Tsai argues that an issue in determining reliability in Service Oriented Systems is how far back should one accept historic reliability information[Tsa05]. He also states that atomic services should have been tested by their authors who should also provide a set of test cases, test oracles and a reliability level. A useful suggestion is that any automatic code generated to create complex services from simpler ones using orchestrations or choreography[9] should generate instrumentation code within those software composition scripts.

Another paper published by different collaborators with Tsai[TEC08] is largely focused on evaluating the reliability of automatically generated service compositions, but makes the point that reliability data from composition execution can be used to recalibrate reliability models and rank or re-rank compositions, services etc. It also suggests that every party in contributing into a service composition can contribute test cases, test scripts and evaluation mechanisms.

Wang et al[WBZC09] offer a hierarchical reliability model for SOA systems in which the system reliability is calculated using the reliabilities of services, fault tolerance mechanisms surrounding them, data and service composition specifications. Compositions are modelled using Discrete Time Markov Chains.

LLAMA[PLZ+08] is a service process monitor, run-time manager and configuration tool. It does not calculate reliabilities, but collects evidence about Quality of Service (even for foreign services) and diagnoses causes of delays through a light weight (on the system) Baysian network and ESB extensions, a similar system might be used to monitor service invocations in an SOA for success or failure.

The notable contributions in this area are the concept of voting as a test oracle for reliability testing new components, incorporating logging of success and failure in automatically generated compositions and designing a system where fault tolerance mechanisms are incorporated in the service architecture and reliability calculations.

[9] See section 3.

3 Orchestration vs. Choreography

A complete process for assessing and guaranteeing reliability of an SOA must deal with service composition. This section, therefore, goes into some detail about service composition paradigms so that explanations about reliability of software composed from other services have a solid foundation.

A major feature of SOA is that it aspires to creating new services by composing existing services, either atomic ones (those that are a single piece of code) or those that are themselves implemented as a composition of lower level services. The objective is not just to reuse code and thus reduce production effort, or to have a canonical set of code managing each type of data — although these are benefits of such an approach; but also to allow the quick construction of new applications using these services that were not envisaged at system design time. The advantages for Defence here in terms of responsiveness in a battle situation are apparent. Generating new services quickly is in some ways antithetical to providing a certifiably reliable system, so this paper will concentrate its efforts in the area of composed applications on how to quantify and improve the reliability of such constructions on the assumption that these compositions can and will be reused.

Service creation by composition is generally broken into two methods which are sometimes combined, they are Orchestration and Choreography. Both methods seek to combine multiple existing services into a coherent larger service / business process. There is not strict agreement on their exact definitions, and they are often described in contrast to each other.

Peltz[Pel03b] gives the following definitions:

Orchestration Refers to an executable business process that may interact with both internal and external Web services. Orchestration describes how Web services can interact at the message level, including the business logic and execution order of the interactions. These interactions may span applications and/or organisations, and result in a long-lived, transactional process. With orchestration, the process is always controlled from the perspective of one of the business parties.

Choreography More collaborative in nature, where each party involved in the process describes the part they play in the interaction. Choreography tracks the sequence of messages that may involve multiple parties and multiple sources. It is associated with the public message exchanges that occur between multiple Web services.

Orchestration differs from choreography in that it describes a process flow between services, controlled by a single party. More collaborative in nature, choreography tracks the sequence of messages involving multiple parties, where no one party truly "owns" the conversation.

However, many authors and practitioners would see orchestration as something to be used primarily within a single organisation. Services created by these methods are embodied in a language file that specifies the underlying service interactions. Given the above definitions, it is possible to see how languages specifying orchestrations are generally procedural whereas languages describing choreographies are generally declarative with the processes

involved responding in an event driven manner to calls made on them. The following quotes from various authors give a wider perspective on the differences between the two broad ways both methods seek to combine multiple existing services into a coherent larger service / business process:

> the point of view of orchestration languages indeed, is always the orchestrator which is the center of the system through which all the information pass[sic].[BGLZ05]

> Orchestration describes how services interact at the message level, including the business logic and execution order of interactions under the control of a single end point[PTDL07].

> [An] analogy is the model of control for the flow of traffic at an intersection. In this case, orchestration would involve a traffic light (controlling when each vehicle gets to cross), while choreography would compare to a roundabout, where there is no central control; there are only some general rules specifying that vehicles approaching the intersection have to wait until there is room to enter the circle [in the correct direction] [Jos07].

> Choreography tracks the message sequences among multiple parties and sources — typically the public message exchanges that occur between Web services —An ... rather than a specific business process that a single party executes[Pel03a].

> choreography depends highly on two things: collaboration and the specification of some general rules so that the collaboration doesn't lead to chaos[PTDL07].

Another way of viewing this is that specifying an orchestration is much like writing your own program, you specify all the operations and the sequential or parallel operations including calls to other services/APIs — orchestration specifies a single (if composite) service and the specification is a single piece of generally executable code; whereas specifying a choreography is like defining a protocol, each party specifies a set of acceptable peers and the stimulus/response behaviours they themselves will undertake, and a corresponding set of allowable interactions and termination or error conditions that may arise based on data. The choreography is the combination of all these individual specifications and represents the "God's eye view" of what is going on. Specifying the behaviour of many services collectively constitutes a larger service, whereas each peer in the choreography is only aware of their small role. Compared to multi-agent systems, choreography is generally less ambitious, usually a single peer's choreography is focused on a very discrete task, and the service endpoints comprising a choreography are generally fixed to a particular host rather than wandering around the system.

Orchestrations are generally executed as a script in response to a call on the service it defines on the host where that script is published as a service. Choreography descriptions need to be interpreted by a program to generate event response automations. The set

of these automations run on each host participating in a choreography, the service interface simply receives the first event in the protocol chain. Note that one fragment of a choreography may, in fact, participate in other choreographies simultaneously. Similarly services can concurrently participate in multiple orchestrations. This leads to the issues of languages for creating orchestrations and choreographies.

3.1 Languages for Orchestration and Choreography

Many languages have been proposed for Orchestration and Choreography or to support them, we cover only some of the most prominent here.

Web Services Description Language (WSDL) is a language that allows programmers to describe the interfaces to web services using an XML format. This XML may then be used to populate service lookup service, or otherwise be scanned by programs seeking to automatically build service compositions. Version 2.0, which became a W3C recommendation in 2007, is considerably different from 1.1 and specifies the *service* - the set of system functions that are exported for access via web protocols; the *endpoint* or address of each service; the *binding*[10] or interaction pattern (e.g. Remote Procedure Call) that it offers; the *interface* or definition of the set of operations that can be performed and what messages are used; the *operations* or method signatures for each "call"; and finally the *types* of data sent and received described by XML schema notation[WSD11].

The Business Process Execution Language for Web Services (BPEL4WS) or just BPEL was designed by BEA, IBM, SAP and Siebel Systems to model the behaviour of Web Services in a Business Process interaction. It is generally regarded as both an orchestration language and a choreography language depending on the level of its use. Its syntax is XML based, making it hard for humans to read or write. WS-BPEL 2.0 is a OASIS (Organization for the Advancement of Structured Information Standards) standard as of 2007[OAS07]. Some aspects of BPEL relate to orchestration while other aspects relate to choreography. *Executable processes* in BPEL reference WSDL signatures of existing services to build private work flows using a common overseer - the BPEL script being executed (orchestration) whereas *abstract processes* describe public message exchanges between executable processes (choreography). Language constructs allow for looping and branching based on data and also allow transactions (including transaction scope) and exception handling to be specified as well[Pel03a].

Other languages which seemed to have lost momentum are WSCI the Web Services Choreography Interface which again provides a collaboration extension to WSDL and the Business Process Management Language (BPML) which is an orchestration language.

WS-CDL or Choreography Description language, like the abstract processes in BPEL is not directly executable. This language has been proposed by W3C, while some in the industry are committed to BPMN the Business Process Model and Notation which is graphic notation for modelling business processes but which can't be easily mechanically translated to or from BPEL[BPM11].

[10]WSDL uses binding to mean the form of interaction pattern, in this paper we use the term in its more common computing usage of resolving a name to an entity such as a service, variable, communication endpoint or address.

To create composed services in a working SOA system, there is a need to combine local services into orchestrations and / or mutually enact the local services described in choreographies and check their overall validity. Researchers such as Diaz Et. Al. are active in this area and advocate deriving a timed automata from the WS-CDL description of a composition, validating the timed automata with a model checking engine, and then deriving WS-BPEL code from the timed automata[DCP+06].

While both orchestration and choreography in the most general case may want to compose services in sequence or in parallel, the languages performing the composition would achieve this in different ways. An orchestrating process would call other services either in sequence or asynchronously but in so doing the called services would be subordinate to the orchestrating process. In fact it would be up to the orchestrating process to take the results from one service call and pass them on to others in the sequence. Conversely, in a choreography, all services described by language fragments that collectively constitute the choreography would be instantiated in parallel. Whether the flow of execution for a particular call to that choreographed service occurs sequentially or with parallel execution would depend on the behaviour of the component services. The only way for a single call to a choreographed service to result in multiple threads of execution would be for some component service of the choreography to make multiple asynchronous calls to other peers in the choreography.

In general, orchestration languages (or parts thereof) tend to be executable while choreography languages (or parts thereof) tend to be declarative - they say what has to be done without explicitly saying how it should be done. There is still considerable dissatisfaction with the state of the art in these languages. For instance: "This sharp distinction between orchestration and choreography is rather artificial and the consensus is that these would coalesce in a single language and environment[PTDL07]." Many authors point out that BPEL and other popular orchestration/choreography languages lack the underlying mathematical rigour to do a full correctness analysis on the protocols being described including aspects of timing, channel allocation and other issues. Indeed several authors have proposed their own extensions to such languages in order to facilitate this goal[CYZQ08, KWH07], others point out that not all the choreography models embodied in these languages are translatable to a set of local implementations[JHG07].

While other implementations of SOA exist, such as ones using message queues or other publish subscribe architectures such as Data Distribution Service (DDS) [OMG] these don't come with their own orchestration and choreography languages.

3.2 ESB and Service lookup

All services in an SOA have to be identified and located before they can be correctly addressed through the ESB and used.

> The main challenge of service discovery is using automated means to accurately discover services with minimal user involvement. This requires explicating the semantics of both the service provider and requester... Achieving automated service discovery requires explicitly stating requesters' needs —

> most likely as goals that correspond to the description of desired services — in some formal request language.[PTDL07]

For us, our major concern with respect to reliability will be at what stage services are bound to choreography or orchestration specifications. There are a spectrum of possibilities but the extreme versions of early and late binding are:

- The implementation, version and indeed which instance of perhaps replicated copies of each web service is hard wired into the script so that there is no flexibility at run-time over which service is called (very early binding) and,
- For every execution of a choreography or orchestration script, a potentially different service instance is selected dynamically to fulfil the role of the service described in the script (very late binding).

Some examples will occur midway through this spectrum, where some aspects of the service chosen are fixed e.g. implementation or service location (local or otherwise) while the other aspects are chosen dynamically. Hauzeur[Hau86] gives good examples about how the rigidity of a naming scheme that locks down component selection can reduce the possibilities of what a system can achieve. As SOA experience and practice matures, common service lookups will become more sophisticated and we can expect more runtime choice as to which service a particular run of a composed service binds to. This becomes an issues in determining SOA reliability because the more freedom that exists in which services are bound into a choreography, the less frequently that particular service instantiation will have been tested in practice. The past evidence available about composed service reliability becomes more scattered and therefore less statistically significant when more dynamic freedom exists in choosing component services for each run of the composed service.

While this paper does not concern itself with the general details of service conformance evaluation involved in selecting a suitable instance of a service for an execution, we will be advocating an approach between these extremes which still allows some run-time flexibility but attempts to maximise the reuse of proven service instance combinations and hence maximises the reliability of the service composition so formed.

3.3 Other service composition issues affecting reliability

Peltz[Pel03a] states that "asynchronous service invocation is vital to achieving the reliability and scalability that today's IT environments require." This means that any assessment of functional reliability for SOA must be able to cope with parallel processes. He also states that "The process architecture must also provide a way to manage exceptions and transactional[sic] integrity." Due to the transaction mix many transactions may have to fail, roll-back and be retried. Such outcomes are the result of normal resource contention, and while they should be minimised they should not be regarded as a failure when assessing reliability of the composition that initiated the transaction.

Other important issues in composing services includes the need to handle multiple versions of services as they are debugged or upgraded to introduce new features. While

Wang and Cui have given a model for determining which other processes in a system may be affected by such changes[WC10], we are interested in how this will affect their reliability and so wish to develop procedures or software components to perform such updates in a controlled manner that maximises reliability during the change over.

We do not consider dynamically generated service composition scripts in this paper. Automatically deriving such compositions is still a significant and open area of research e.g. "Despite all these efforts, composition of web services still remains a highly complex task, and automatic composition of web services is a critical problem[MA07].", and even now there does not appear to be any strong agreement on a way forward. There are already significant challenges in providing reliability assessment for human crafted service compositions, so we will confine our efforts in this paper to that task which is a prerequisite for the more complex problem as well.

4 Applying Software Reliability Engineering to SOA

Just because SOA is intended to allow the rapid development of new applications from existing components through service composition, there is no reason to abandon good software engineering practice. In fact the more agile one needs to be, the more one should be able to rely on a solid Software Engineering foundation to enable such agility. This applies to Software Reliability Engineering as well: development cannot be agile if it gets bogged down in tracking down the distant and obscure root causes of many bugs.

In this section we go through the sequence of steps we see as most useful in developing a component or composition for an SOA (which for the time being we assume is itself reasonably reliable) and see how the various processes expounded in section 2 apply specifically to the SOA context. We are fortunate to have a variety of approaches which we can mix and match in order to obtain some of the most suitable solutions.

Specification A specification of the service is invaluable to clarify development and to be clear about a service's behaviours. Box specifications such as those recommended in Cleanroom Software Engineering allow for very clear indications of expected outputs for each input and as such lay a foundation for automatically generating test data. Neither Tsai et al's weighted voting approach, nor Wang et al's hierarchical reliability model, nor LLAMA focus on specification.

Usage Model / Operational Profile Both Cleanroom and Software Reliability Engineering insist on a means of determining which operations are most used in a system in order to ensure that testing covers the likely actual usage of the system. Cleanroom's Usage Model is its usage Markov chain as described in section 2.1 whereas Musa's Software Reliability Engineering uses Operational Profiles. Markov chain models together with Cleanroom type box specifications can be used to automatically generate test data with the correct coverage for expected usage.

We note that it is useful to create a formal specification and usage model such as those advocated by Musa or the Cleanroom methodology even for software that

has come from somewhere else or is legacy code. As odd as this sounds, doing so provides a way to automatically generate tests for the foreign or legacy software as it is *now* intended to be used. As pointed out earlier different faults may be revealed by different usage and prior testing is advisable. Likewise, it is advisable to create an updated usage model for a component for its new use when it is "conscripted" into a new service composition. A good hierarchical SOA monitoring system will be able to log the calls to the newly constructed composed service as well as to its existing constituent components so that an updated overall usage model can be calculated for the component (by multiplying through the macro composite service usages and the per composite service usage models). A specification and usage model for bought software can also provide a way to do acceptance testing of the Software.

Creation of a usage model or operational profile is also a good time to check if any operations can be eliminated to simplify the system and so reduce the chances of residual faults.

Determine reliability objectives This may be problematic for components that are being built bottom up, rather than from true user requirements. A typical example of this is designing service infrastructure, e.g. a database service, where the need for some services is obvious in the SOA ecosystem but it is less obvious what exact form each service will take and how precisely different higher level services and applications will use them. However, it is still useful to make an effort here as it can inform when is the earliest time the new component should be released into the SOA ecosystem. Musa offers a simple subtractive process to determine necessary component reliabilities, Cleanroom offers a more mathematically rigorous deduction arising for the usage Markov chains. It is unclear to the author which would be better, and this may well depend on context.

Determine a common unit of measure for reliability This is an interesting question in an SOA and is discussed in section 4.2.

Plan and develop tests If a specification and usage Markov chain has been developed, test generation can be automated saving considerable developer time.

Perform testing This will hopefully be done automatically either using the usage Markov chain type testing, or by the group voting method espoused by Tsai et al. Either way the results should be automatically recorded in the system to accumulate evidence for reliability. It is important that testing include both testing of operations and significant loading levels beyond the expected system load. Regression testing should also be performed.

This means the logging service has to be sophisticated enough not only to log the success or otherwise of the macro composition, but also each of its constituent components recursively. This also requires the logger to have a clear idea about which particular instance of each component service including location and software version was invoked.

Calculate and update reliability levels More so than in static systems, much of the data on reliability of an SOA will come from experience from fielded components. Every in-field use of a service can be considered a test. As such testing progresses,

the reliability levels of any composition being exercised and its components should be updated. This is the only way that reliability data on ad hoc and machine derived service compositions can be accumulated, although estimations of reliability for such service compositions can be derived from their components' existing reliability records. There should be a reliability database that includes both an estimated reliability entry and a certified reliability entry. Exactly when and how data obtained through normal system operation, or through Tsai style group testing should be deemed to certify a reliability is an area for further investigation. These methods of evidence accumulation are essentially uncontrolled. Clearly the production system defines the true rather than extrapolated or expected operational/usage profile, but the question that must be answered is "What is a sufficient sample size?" Specific certification testing as per Cleanroom or Musa's SRE immediately qualifies as a certified level once obtained.

Deploy Certified Software This should usually be the only way that software gets deployed, however in war fighting situations Defence may wish to lower the bar somewhat to allow for agile responses. Even so, some minimum levels of component reliability should be enforced so that the overall system remains stable under the usage conditions for which they were certified.

Verify Deployed Reliability It is important to continue to log successes and failures in the field, not only for debugging and software improvement, but also to ensure that our reliability modelling and decision processes are scrutinised and improved.

As composed services are themselves services, the same advice as above should be applied to them as well. They should have a specification, usage model etc. Even if done after the event of war fighting, they will be useful in the future. They should also be tested as a unit because the script itself can introduce errors on top of otherwise correct services.

4.1 A layered approach to Certifying the SOA

In systems as large as a Defence wide SOA, it is neither practical nor economic to certify the system as a whole. The task would be enormous, error prone and the results unreliable. A better and tractable approach is to view the Defence wide SOA as a set of separate interacting applications, which intersect each other as they use common services and infrastructure. By treating these services and applications as a layered system, with layers defined by the direction of calls between them, we can gradually certify the reliability of each service or application in the system according to the known reliabilities of the services they themselves call. In this way Defence also has a more useful reliability measure because it will give a reliability for the specific applications and services that they seek to use in any deployment, making decisions clearer.

4.2 Certifying from the core out

Figure 3 shows a generic SOA System including its innermost element, the operating system at the centre. This figure is a logical representation, and the SOA itself would

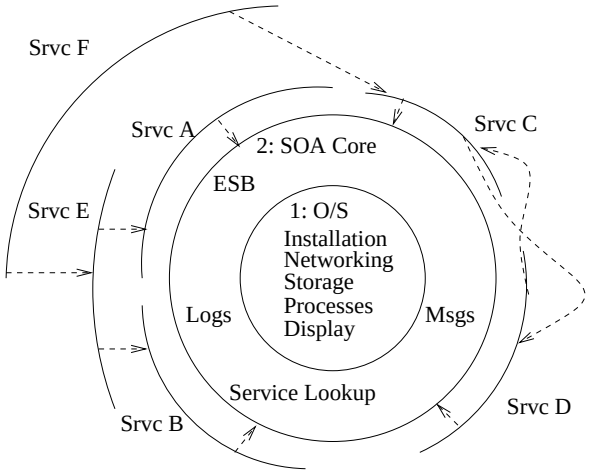

Figure 3: *Layered Services in an SOA*

consist of many nodes, each with its own operating system. In the figure, each layer shown represents the union of that layer across all nodes of the physical SOA. Ideally the operating system should be as minimalist as possible to maximise reliability by reducing the amount of code that may contain faults. For an SOA it would necessarily contain installation, networking, process (and within node inter-process communication/calling) components and optionally display and storage components which would only be included on nodes that required them. In figure 3 this constitutes layer 1 and should be the first element of the SOA to be certified.

Following that, the services comprising the SOA Core can be added to the system. One possible minimal set of these would be the messaging/ESB, logging and service lookup functions. This is layer 2 of the system and its reliability can be certified using the results obtained from certifying the operating system.

Together, layers 1 and 2 constitute the core infrastructure of the Defence SOA and should be certified separately because it is fundamental to system operation and will need to be rolled out first in any case. It is very important that a software specification and usage model[11] be developed for the core functionality, even though at first this will only be an informed estimate. This will ensure the highest initial level of reliability for the core infrastructure by focusing development effort on the most used parts of the core infrastructure. The operations used by the layer two functionality will inform which calls in the operating system are the most frequently used. Clearly the usage model for the core infrastructure will change over time as more applications/systems are added to the infrastructure and estimates of usage are revised with real data. However, this is no excuse for not developing an initial usage model. As the particular SOA ecosystem develops, the proportion of the core functionality usage model which is derived from estimates as opposed to measurements and past experience will continue to fall.

The measure of software reliability needs to be compatible among the core infrastructure and the systems built upon it. There are basically two possible types of measures:

[11] Usage Markov chain or Operational Profile

- Failures per time period (e.g. failures/hour)

- Failures per top level operation invocation/footnoteTop level for the application, or top level for the infrastructure.

While core infrastructure should be continuously operational, other systems may be used sporadically even if continuously turned on. For this reason the Author currently conjectures that measuring SOA reliability in failures per top level invocation will give a more meaningful measure against which to define the require reliability as we are generally uninterested in failures that might occur when we are not actively using the system. Moreover, this measure scales well when we distinguish between reliability during non-use, off-peak and peak usage times. This is particularly true for different modes of system operation. A change in system mode may result not only in a different intensity of invocation, but in a different operational profile as various tasks in the overall SOA System are reprioritised.

4.2.1 Certifying individual Services

Each individual service on a Defence system developed on the SOA infrastructure should be tested as a whole using its own specification and operational profile. In figure 3, we can see what is effectively layer 3 consisting of services A, B, C and D. This layer is not a closed circle because its components are optional. Because of this, layer three would not be certified as a whole, but each of its services would be certified individually using the known reliability of the SOA core. Note that since services C and D are not independent but call each other, they would need to be certified as a group yielding a common reliability. This provides extra motivation for separating concerns into layers in an SOA System recognising that sometimes software functions are intertwined although best programmed separately. In this figure, service E represents a fourth layer and its reliability can be certified once services A and B have been certified. Similarly it is logical to certify service F after services E and C/D are certified.

In effect services should be certified from the core out, or in the case of remote services all services called by the service in question should be certified before that service itself is certified. This ordering maximises the efficiency of software development by ensuring that each service is being built on a solid foundation and developers are largely debugging problems in their own code, not others'.

As service reliability and behaviour may be highly dependent on network connectivity in an operational system, reliability may need to be calculated both for the case where all parts of the individual system are local to the caller and for the case where some functions are performed remotely. This would depend on the nature of the service and the possibilities for meaningful disconnected operation. Any failures during certification must distinguish whether the failure occurred due to a connectivity problem, or a fault intrinsic to one of the services involved.

4.3 How we might estimate reliability of Service Compositions

Composed reliability can be estimated by applying Markov usage models of the service composition, treating each called service as a component. Like all other systems, reliability of service compositions can only be certified by applying certification testing using a valid operational profile. Evidence of in field usage or group based testing can be accumulated to provide an estimated reliability, but as stated before when this should be regarded as a large enough sample to call it statistically representative of the real operational profile and thus qualify as certification is another question.

In section 4.2.1 we advocated an ordering for attempting service reliability certification. This ordering also enables estimation of the maximum reliability of a new service given the known reliabilities of the services it depends on using the Markov chain approach described in section 2.1. Such an estimation using only the expected calling frequencies of existing services can inform the developer whether there is a decent chance of making the new service reliable enough or whether development efforts are currently best spent increasing the reliability of services it depends on. In such a case it will also show which of those services needs to be improved first.

4.4 Performing Upgrades

In an operating SOA good governance, both human and embodied in system policies, must be used to ensure that upgrades of components and services are both possible and keep the system relatively stable during the upgrade. The following protocol should be applied for upgrading services:

- Replacement components have to meet minimum reliability levels through automated testing before they are eligible to begin testing in live systems.
- A replacement component should be group tested in the Tsai style in live systems and shown to approach the reliability levels of the current component before it is eligible as a replacement.
- The replacement component is then registered and publicly published in the White, Yellow and Green pages and is chosen by default over the previous version once it has passed these tests. This allows for rapid accumulation of reliability data.
- If the node is running in War/High Reliability mode, the component will not be selected for a composition unless the user of the composition script specifically requests that the newest published components be used or accumulated data shows that it meets or exceeds the established reliability levels of the previous version.

This process must be supported by appropriate logging[Pil12] so that field evidence is accumulated about the replacement component.

4.5 Certifying Legacy Systems as Components

The potential to incorporate legacy systems as wrapped[12] services in an SOA is seen as having several potential benefits such as:

- Extending the operational life of a legacy system
- Extending the accessibility of a legacy system by making it widely available
- Extending or altering the functionality of a legacy system by incorporating it within a larger service.

Most but not all legacy systems have been operational for a significant time. Usually they have come to be used for purposes not fully envisaged at the time of their construction. Moreover if they are being incorporated into an SOA they are likely to be used in a different way than they have been either in frequency of use or in the calling pattern they are subjected to.

Like all reliability measurements discussed in this paper, the reliability measurement and certification of a wrapped legacy component must be done in with respect to a specification and a usage model. With legacy systems the system's original specification may no longer exist, may not be available, or a written specification may never have been produced. It is also unlikely that a usage model was ever produced.

The solution is to write a new specification for the legacy component, not for what the legacy system has done, but for what the legacy system will do - and is expected to do - in the SOA. At the same time a usage model must be developed for the intended/expected usage of the legacy system as a component.

Not only will this ensure an accurate reliability measure and certification for the legacy system's **current** purpose, it is highly likely to pick up any misunderstandings about the function of what is often effectively a black-box system. These misunderstandings may become evident either during the specification process as rigour clarifies thinking, or they may manifest as errors during the certification testing.

Note that the newly developed specification for the legacy system being wrapped does not need to specify all its native behaviours, only the input and output behaviours expected from the portion of the the legacy system's interface that is being wrapped by the SOA service. As such, the work required to produce such a specification is limited and feasible.

This specification and the associated usage model of the wrapped service can be used to automatically generate a test suite which can be used to measure and certify the systems actual reliability for its **current** purposes within the SOA.

As counter-intuitive as it may first appear, the above reasons show that specifying a legacy system after it already exists makes perfect sense.

[12] Wrapped in interface code to facilitate their incorporation into the SOA so as to appear as a service.

5 Implications for ADF procurement

In order to establish and maintain the reliability of SOA based systems delivered to the ADF, Defence should accept incremental delivery of the system functionality but with strict requirements on the reliability of the entire system thus far delivered at each increment. This contrasts with traditional Defence acquisition and ensures that SOA reliability starts high and remains high. Keeping reliability high speeds development and allows portions of functionality to be deployed earlier than would otherwise be possible. It also has the distinct advantage of increasing return on investment because returns begin sooner.

Certification testing against agreed operational profiles, specifications, and confidence levels provides a very clear method for suppliers to know when to offer deliverables to the ADF, and for the ADF to determine whether to accept the deliverables.

Operating systems are a special case here. The ADF is somewhat constrained by the limited number of operating systems available. However a usage model can be developed for the smaller subset of an operating system called on by an SOA (process control, communications and networking, security) and develop an Operational Profile / Usage Model for this subsection of the Operating System's API. This can be certified to a measured level of reliability and this measure can be used in conjunction with known malware and security threats to either choose between operating systems or to decide when a new version of an operating system is mature enough to allow into the SOA.

6 Recommendations to enable SOA SRE

Software Reliability Engineering is based on a mature set of theories that have been shown in practice to produce significant improvements in software reliability at affordable cost in industry software development and in the U.S. DoD[DPC03]. As such, Australian Defence should adopt a form of SRE for its SOA systems procurement, both to guide development of bespoke software and as a test for acceptance for bespoke and off-the-shelf software. The following items describe how this can be done:

- One of the clearest paths to deployment is to implement functionality incrementally starting with the core functions and only release each increment when the reliability of that release set is certified high enough for deployment. This path ensures system stability and earliest possible access to functionality on an incremental basis. It also assists projects to accurately report their progress by allowing them to say that a particular percentage of the project's (sub) functions are certified to a given reliability.
- The deliverables for components should include their specifications and some form of usage model. Ideally these would be in the form of Box specifications and Markov chain usage models so they can be used to automatically generate test suites.
- Services that modify persistent data, rather than simply returning a result, such as updating a database should include a testing mode in which either the updates do not occur or the data is stored in an separate testing database. This allows the

service to participate in testing without contaminating production data. Another way of testing services that have side effects on real infrastructure would be to stand up a separate test instance of the service. This would have the advantage of not altering the code of the service being tested at all but would mean the tests would be conducted without the load from other users of the service so interaction effects would not show up unless included in the tests themselves.

- That Markov-chain usage based testing be adopted in Defence for SOA and other systems. For systems delivered to the US DoD, this type of testing allowed automated test oracles to be developed allowing automated testing, and caused a ten fold reduction in defects per 1000 lines of code[DPC03].

- As coalition operations move towards interoperability through sharing SOAs, it would be advantageous to agree early on standard interfaces for each type of service including WSDL interfaces and calling grammars. This will allow maximum usage of each country's service by others and allow the services to be substitutable for testing and war fighting purposes. The same is of course true for all foreign APIs, but there is perhaps more room for early negotiation on common data structures and API structure with Defence's Coalition partners.

- Some mechanism to provide data provenance should be a core part of the system specification. This is important for ensuring faults are not caused by bad or inappropriate use of data. It is particularly important to be able to correct databases by expunging data that has been inserted into the database by a faulty service.

- As part of SOA governance, the software repository should also include the following data for all services:

  - Specifications of the service.
  - Usage models for all contexts in which the service is known to be used. (These may be lodged by parties other than the software developers.)
  - Provenance of the service: who wrote it, what techniques were used, how the specification was validated etc.
  - For legacy GOTS and COTS software, a specification describing expected behaviour of the APIs of the software used should still be written (even after the event) to provide a foundation for testing along with the associated usage model.

- A key motivation given by CIOG for adopting SOA was to enable software reuse and by implication to disaggregate applications so that their component services could be reused to build other applications rather than each application reinventing the wheel. To facilitate such reuse, application developers and automated processes that build orchestrations and choreographies must be confident of each component's behaviour and reliability and hence that they are "fit for purpose" for a particular usage. The specifications mentioned, and to some extent the usage model provides a "proof of behaviour" under expected circumstances. To provide proof of reliability, the SOA system/governance must maintain a database of reliability levels for each version of a component. Initially this level may be an estimate based on past performance

of the software supplier or the software development techniques used, later as more tests are performed against the usage model and specification this reliability can be measured within the given usage context. Initial testing and limited deployment in the new usage context has to rely on previous usage contexts as an approximation of reliability. Once testing or operational reliability data specific to the new usage context can be accumulated there is a corresponding increase in the confidence we have in a particular reliability level for that component version and usage.

- SOA governance should include minimum reliability levels for different types of services in its SOA systems. It can be hard for programmers to estimate the needed reliability for whole systems and guidelines would avoid a lot of indecision. There should be a variety of levels for each type of service that should apply as it progresses from development to test, to transfer onto production machines and finally into deployment and advertising as an available service in the White, Yellow and Green Pages services for the SOA.

- A service placed in the lookup pages should be able to be marked experimental so it can be fielded for testing, but not accidentally chosen as a subcomponent until it has been certified. This allows field testing as a shadow service where every call is routed to both the existing service and its replacement and outputs can be compared to validate the new service.

- Defence SOA nodes should have at least three system modes:

  Test mode The hardware is available for running as yet unreliable, lightly tested components for the purposes of testing. It is acknowledged that there is a possibility that the testing may bring down the local part of the SOA infrastructure.

  Normal mode Only software/services that have passed minimum reliability tests are available for execution.

  War or High Reliability mode Only software/services that have passed more stringent software certification or which have been authorised to run to obtain a war fighting advantage despite their reliability risk are available for execution.

  Software running on a test node should be allowed to call software on a normal mode during the course of its testing. Allowing these modes to exist simultaneously on different nodes in the broad ADF SOA infrastructure provides a path for new Services and SOA components to climb the reliability ladder until they are certified fit to deploy.

7 Conclusions

At this point we can answer the questions we posed at the beginning of this paper:

- The integrated grid of Defence SOA platforms may be sprawling; how can one talk about reliability for such a system? We break it down as described above and certify the core infrastructure and each significant system built on top of it separately. Effectively, an application system is certified from the SOA core outwards as each

service is certified. The network is treated as just another component of the system, it has a separately measurable reliability.

- How can one be sure a Defence system built on SOA is reliable enough to deploy? By using good software reliability engineering practices and only deploying reliability certified systems.

- Some components of Defence SOAs will be wrapped legacy systems, how can and should these be certified? The best way to certify these is to develop a **new** specification and operational profile for them that corresponds to our current intended use. As counter-intuitive as creating a new specification for an old and most likely unmaintainable existing piece of software may seem, this method importantly ensures that its reliability is known for its new uses as well as its old ones. This technique is a significant contribution to our capacity to use legacy systems with confidence.

- How can one deal with components built and or maintained by others and still get a reliable system? Clearly if a component is of too low reliability it should be rejected. However, with software reliability engineering practices we can determine its actual reliability for our user as described for legacy systems, and we can compare that with a Markov usage model analysis of component reliabilities needed to meet the reliability objective of the larger system. The component should only be accepted if it meets or exceeds that requirement.

- What can be said about the reliability of services created by composing other services? The reliability of composed services can be measured and certified just like any other system.

- How do multiple versions of the same component affect reliability and the calculation and certification of reliability? Each version will have its own inherent reliability, and its own level of currently certified or evidenced reliability. If the versions are independently authored, it is simply a matter of choosing the one with higher reliability; however, most often the versions will be related with the newer version offering some feature enhancement or bug fix. In the latter case the objective is to raise the newer version to an acceptable level of certified reliability as soon as possible. Section 4.4 describes a systematic means to do this using the older component as a test comparison, and to preferentially select the newer component where possible to accumulate statistical evidence on its reliability for deployment.

- What can one say about the reliability of compositions that include foreign services? These should be analysed in the normal way. The fact that the service is foreign does not affect its reliability. It does however affect its time to correction if a bug is discovered by *us* during testing. As such it is important to have the ability to submit good bug reports[Pil12] and to have an agreement with the service supplier about correcting faults in a timely manner.

This paper has answered many questions about ensuring an adequate level of system and application reliability for the Defence SOA. The ability to do so is strongly predicated on adopting proven software reliability engineering techniques and it is highly recommended that Defence adopt such techniques both in its software development and software procurement policies.

While this paper has outlined how existing SRE techniques would translate to SOA, detailed elaboration of them will be necessary to create a methodology with which to pursue SRE in SOA development and acquisition.

Acknowledgements

The author would like to thank Steven Wark and Derek Henderson for their helpful review comments.

References

BGLZ05. Mario Bravetti, Claudio Guidi, Roberto Lucchi, and Gianluigi Zavattaro. Supporting e-commerce systems formalization with choreography languages. In *Proceedings of the 2005 ACM symposium on Applied computing*, SAC '05, pages 831–835, New York, NY, USA, 2005. ACM.

BPM11. Wikipedia entry for Business Process Model and Notation, 2011. http://en.wikipedia.org/wiki/BPMN Viewed 2011-05-17.

CIO10. CIOG. Single Information Environment (SIE): Architectural Intent 2010. Technical Report DPS: DEC013-09, Commonwealth of Australia, Department of Defence, May 2010.

CoA11. Department of Defence Commonwealth of Australia. Chief information officer group instruction no. 1/2011. Departmental dissemination., May 2011.

CYZQ08. Chao Cai, Hongli Yang, Xiangpeng Zhao, and Zongyan Qiu. A formal model for channel passing in web service composition. In *Services Computing, 2008. SCC '08. IEEE International Conference on*, volume 2, pages 495 –496, july 2008.

DCP+06. G. Diaz, M.E. Cambronero, J.J. Pardo, V. Valero, and F. Cuartero. Automatic generation of correct web services choreographies and orchestrations with model checking techniques. In *Telecommunications, 2006. AICT-ICIW '06. International Conference on Internet and Web Applications and Services/Advanced International Conference on*, page 186, feb. 2006. Very good high level summary of WS-CDL.

DPC03. S Dalal, J Poore, and M Cohen. *Innovations in Software Engineering for Defense Systems*. The National Academies Press, Washington, D.C., 2003. http://www.nap.edu/catalog/10809.html.

Fis09. Neal A. Fishman. *Viral Data in SOA: An Enterprise Pandemic*. IBM Press, 2009. ISBN-13: 978-0-13-700180-4.

Hau86. Bernard M. Hauzeur. A model for naming, addressing and routing. *ACM Trans. Inf. Syst.*, 4:293–311, December 1986.

JHG07. Li Jing, Zhu Huibiao, and Pu Geguang. Conformance validation between choreography and orchestration. In *Theoretical Aspects of Software Engineering, 2007. TASE '07. First Joint IEEE/IFIP Symposium on*, pages 473 –482, june 2007.

Jos07. Nicolai M. Josuttis. *SOA in Practice*. /Theory/In/Practice. O'Reilly, 1005 Gravenstein Highway North, Sebastopol, CA 95472, 2007. ISBN-13: 978-0-596-52955-0.

KWH07. Zuling Kang, Hongbing Wang, and Patrick C.K. Hung. Ws-cdl+: An extended ws-cdl execution engine for web service collaboration. In *Web Services, 2007. ICWS 2007. IEEE International Conference on*, pages 928 –935, july 2007.

MA07. Sun Meng and Farhad Arbab. Web services choreography and orchestration in Reo and constraint automata. In *Proceedings of the 2007 ACM symposium on Applied computing*, SAC '07, pages 346–353, New York, NY, USA, 2007. ACM.

MIO87. J.D. Musa, A. Iannino, and K. Okumoto. *Software Reliability: Measurement, Prediction, Application*. McGraw-Hill Book Company, 1987. ISBN 0-07-044093-X.

Mus04. John D. Musa. *Software Reliability Engineering: More Reliable Software Faster and Cheaper*. Author House, 1663 Liberty Drv., Suite 200, Bloomington, Indiana 47403, 2nd edition, 2004. ISBN: 1-4184-9387-2 Order from http://www.authorhouse.com.

Nik02. A. P. Nikora. Casre 3.0 users guide. Program and Users Guide downloadable from http://www.openchannelfoundation.org/orders/index.php?group_id=250 Viewed Nov 2011, 2002.

NL05. Eric Newcomer and Greg Lomow. *Understanding SOA with Web Services*. Addison-Wesley, Boston, MA, 2005.

Nyg07. Michael T. Nygard. *Release It! Design and Deploy Production-Ready Software*. Pragmatic Programmers LLC, http://www.pragmaticprogrammer.com, 2007. ISBN: 978-0-9787-3921-8.

Oas06. Reference model for service oriented architecture, October 2006. http://docs.oasis-open.org/soa-rm/v1.0/.

OAS07. Web services business process execution language version 2.0, April 2007. OASIS Standard http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html.

OMG. Omg data distribution service portal. Website. "Viewed August 2012".

Pel03a. C. Peltz. Web services orchestration and choreography. *Computer*, 36(10):46 – 52, Oct. 2003.

Pel03b. Chris Peltz. Web services orchestration and choreography, June 2003. http://soa.sys-con.com/node/39800 viewed Dec. 2011.

Pil12. Michael Pilling. Debugging and logging services for defence service oriented architectures. Technical Report DSTO-TR-2664, DSTO, Department of Defence, PO Box 1500, Edinburgh SA 5111, Australia, 2012.

PLZ+08. Mark Panahi, Kwei-Jay Lin, Yue Zhang, Soo-Ho Chang, Jing Zhang, and Leonardo Varela. The llama middleware support for accountable service-oriented architecture. In *Proceedings of the 6th International Conference on Service-Oriented Computing*, ICSOC '08, pages 180–194, Berlin, Heidelberg, 2008. Springer-Verlag.

PMM93. J.H. Poore, Harlan D. Mills, and David Mutchler. Planning and certifying software system reliability. *IEEE Software*, 10(1):88–99, January 1993.

PSRF09. Larry Pizette, Salim Semy, Geoffrey Raines, and Steve Foote. A perspective on emerging industry soa best practices. *CrossTalk The Journal of Defense Software Engineering*, pages 29–31, July / August 2009. http://www.stsc.hill.af.mil.

PTDL07. M.P. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann. Service-oriented computing: State of the art and research challenges. *Computer*, 40(11):38–45, Nov. 2007.

PTLP99. Stacy J. Prowell, Carmen J. Trammell, Richard C. Linger, and Jesse H. Poore. *Cleanroom Software Engineering: Technology and Process*. SEI Series in Software Engineering. Addison-Wesley Professional, 1999. ISBN-10: 0-201-85480-5, ISBN-13: 978-0-201-85480-0.

Smi90. Connie U. Smith. *Performance Engineering of Software Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1990.

SOA06. Wikipedia entry for Service-Oriented Architecture, 2006. http://en.wikipedia.org/wiki/Service-oriented_architecture Viewed 2011-05-17.

SW01. Connie U. Smith and Lloyd G. Williams. *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*. Addison-Wesley Professional, 2001. ISBN-10: 0201722291, ISBN-13: 978-0201722291.

SW04. Connie Smith and Lloyd Williams. *Software Performance Engineering*, chapter 16, pages 343–365. Springer US, 2004. 10.1007/0-306-48738-1_16.

TEC08. Wei-Tek Tsai, Jay Elston, and Yinong Chen. Composing highly reliable service-oriented applications adaptively. In *Proceedings of the 2008 IEEE International Symposium on Service-Oriented System Engineering*, pages 115–122, Washington, DC, USA, 2008. IEEE Computer Society.

Tsa05. W. T. Tsai. Service-oriented system engineering: A new paradigm. In *Proceedings of the IEEE International Workshop on Service-Oriented System Engineering (SOSE)*, pages 3–6, 2005.

TWZ+07. Wei-Tek Tsai, Xiao Wei, Dawei Zhang, Ray Paul, Yinong Chen, and Jen-Yao Chung. A new soa data-provenance framework. In *Autonomous Decentralized Systems, 2007. ISADS '07. Eighth International Symposium on*, pages 105 – 112, march 2007.

TZC+04. W. T. Tsai, D. Zhang, Y. Chen, H. Huang, R. Paul, and N. Liao. A software reliability model for web services. In *The 8th IASTED International Conference on Software Engineering and Applications*, pages 144–149, 2004.

WBZC09. Lijun Wang, Xiaoying Bai, Lizhu Zhou, and Yinong Chen. A hierarchical reliability model of service-based software system. In *Computer Software and Applications Conference, 2009. COMPSAC '09. 33rd Annual IEEE International*, volume 1, pages 199 –208, july 2009.

WC10. Mi Wang and LiZhen Cui. An impact analysis model for distributed web service proces. In *Computer Supported Cooperative Work in Design (CSCWD), 2010 14th International Conference on*, pages 351 –355, april 2010.

WP92. J.A. Whittaker and J.H. Poore. Statistical testing for cleanroom software engineering. In *System Sciences, 1992. Proceedings of the Twenty-Fifth Hawaii International Conference on*, volume ii, pages 428 –436 vol.2, jan 1992.

WSD11. Wikipedia entry for Web Services Description Language, 2011. http://en.wikipedia.org/wiki/Web_Services_Description_Language Viewed 2011-05-17.

DEFENCE SCIENCE AND TECHNOLOGY ORGANISATION DOCUMENT CONTROL DATA	1. CAVEAT/PRIVACY MARKING

2. TITLE	3. SECURITY CLASSIFICATION
Reliability Engineering for Service Oriented Architectures	Document (U) Title (U) Abstract (U)
4. AUTHOR	5. CORPORATE AUTHOR
Michael Pilling	Defence Science and Technology Organisation PO Box 1500 Edinburgh, South Australia 5111, Australia

6a. DSTO NUMBER	6b. AR NUMBER	6c. TYPE OF REPORT	7. DOCUMENT DATE
DSTO–TR–2784	AR-015-477	Technical Report	February 2013

8. FILE NUMBER	9. TASK NUMBER	10. TASK SPONSOR	11. No. OF PAGES	12. No. OF REFS
2012/1082931/1	CDG 07/355	DG Integrated Capability Development	36	41

13. URL OF ELECTRONIC VERSION	14. RELEASE AUTHORITY
`http://www.dsto.defence.gov.au/publications/scientific.php`	Chief, Command, Control, Communications and Intelligence Division

15. SECONDARY RELEASE STATEMENT OF THIS DOCUMENT

Approved for Public Release

OVERSEAS ENQUIRIES OUTSIDE STATED LIMITATIONS SHOULD BE REFERRED THROUGH DOCUMENT EXCHANGE, PO BOX 1500, EDINBURGH, SOUTH AUSTRALIA 5111

16. DELIBERATE ANNOUNCEMENT

No Limitations

17. CITATION IN OTHER DOCUMENTS

No Limitations

18. DSTO RESEARCH LIBRARY THESAURUS

Software Reliability	Certification
Reliability Engineering	Software Maintenance
Software Evaluation	Service Oriented Architecture

19. ABSTRACT

This paper reviews the state of the art in Software Reliability Engineering (SRE), and adapts these methods for use in Service Oriented Architecture (SOA). While some prior work has been done on using SRE for SOA, it is incomplete in terms of whole of development life cycle methodology. We outline how existing complete methodologies would translate to SOA and provide a surprisingly simple way of applying these methods to certify the use of legacy software in SOAs. This paper provides a proof of concept but further work needs to be done to elaborate these methodologies for application in SOA.