

Using Safety Contracts to Identify Regression Tests for Modular Systems

Project Dissertation
Submitted for the Degree of

MSc – Safety Critical Systems Engineering

University of York
Department of Computer Science

Christine Hollinshead
September 2008

Abstract

Modular systems are increasingly being used for both non-safety critical systems and (more recently) safety critical systems. While the modules can be tested/regression tested as independent units, the integration of the modules will create emergent properties that must also be tested/regression tested. Where the system is safety critical, the system must also be accepted (and possibly certified) before use, and after each incremental update. Regression test evidence will form part of the acceptance case for an updated modular system.

To simplify the selection of regression tests, and to save time and resource, a heuristic is often used, with regression test selection being risk based (e.g. focusing testing on areas associated with highest incurred costs following failure), or functionality based (e.g. focusing testing on the most frequently used functions). However, in safety critical systems where integrity must be compellingly argued through a safety case, the use of heuristics may be difficult to justify and defend.

There are parallels between the modular components used within modular systems, and the software components developed using the object-oriented paradigm. Both are designed to be reusable and interchangeable. At the time of usage, the modules and objects (runtime instantiations of classes) provide units of functionality, bounded by interfaces that allow interactions with other components in the system.

Design by Contract has been developed as a means of specifying and implementing object-oriented software components, by defining mutually obliging contracts between the components. To support inheritance, rules have been defined linking inheritance, component inter-changeability and contracts.

This dissertation applies the rules for inheritance and component interchange from within Design by Contract to the problem of identifying regression tests for modular systems. This technique provides a well-grounded method for the identification of regression tests, which could be defensibly argued within a safety case. However, the technique requires that a project be based on the principles of Design by Contract, that all the Client/Server pairs within a system are identified, and significant data management in the form of contract management and contract comparison.

Acknowledgement

Thanks to Dr Mark Nicholson who acted as project supervisor, for his guidance during the writing of this dissertation.

Statement of ethics

The basic ethical principles applied while working on this project were:

- Do no harm
This project has involved two people: my supervisor and myself. Neither of us has come to any harm as a result of our involvement.
- Informed consent
Not applicable (no human subjects other than my supervisor and myself have been involved in this project).
- Confidentiality of data
Not applicable (no information regarding individuals has been collected).

Contents

1	Introduction	6
1.1	Background	6
1.2	Scope	8
1.3	Structure	8
1.4	Dissertation conventions	8
2	Literature Survey	9
2.1	Design by Contract	9
2.2	Design by Contract at the System Level	18
2.3	Modular Systems	21
2.4	Acceptance and incremental acceptance of modular systems	26
2.5	Emergent properties and Design by Contract	30
2.6	Redefining and matching contracts	33
2.7	Finding safety contracts in layered architectures	35
2.8	Testing	37
2.9	Regression Testing	42
2.10	Safety Cases and Goal Structured Notation	45
2.11	Summary of literature survey	48
3	Problem statement and proposed approach	49
3.1	Problem statement	49
3.2	The approach	50
4	The working example	51
4.1	Active steering – an example of a safety critical system	51
4.2	Working definitions	53
4.3	Assumptions	53
5	Safety contracts and regression testing	55
5.1	Integer based safety contracts in a typical safety critical system	55
5.2	Changes to integer based safety contracts	56
5.3	Conclusion	57
6	Supplier regression testing via integer safety contracts	58
6.1	Scenario 1a – no change to Supplier’s safety contract	58
6.2	Scenario 1b – no change to Supplier’s pre-contract	59
6.3	Scenario 1c – weaker Supplier pre-contract	59
6.4	Scenario 1d – stronger Supplier pre-contract	60
6.5	Scenario 1e – complex change to Supplier’s pre-contract	60
6.6	Scenario 1f – no change to Supplier’s post-contract	61
6.7	Scenario 1g – weaker Supplier post-contract	61
6.8	Scenario 1h – stronger Supplier post-contract	61
6.9	Scenario 1i – complex change to Supplier’s post-contract	62
6.10	Combining changes to pre-contracts and post-contracts	62
6.11	Conclusion	63
7	Process to identify regression tests via safety contracts	64
7.1	Step 1 – Identification of the initial test set	64
7.2	Step 2 – Identification of the updated safety contracts	65
7.3	Step 3 – Compare safety contracts to identify regression tests	65
8	GSN Patterns	69
8.1	Generic GSN pattern for regression testing	69
8.2	GSN module for identification of system DSRs and safety contracts	71
8.3	GSN pattern for regression testing via DSRs and safety contracts	74
9	Integer non-functional safety contracts and regression testing	82
9.1	A typical safety critical system – non-functional requirements	82
9.2	Timing constraints	82
9.3	Scenario 2 – timing constraints and safety contracts	83

9.4	SIL constraints	85
9.5	Scenario 3 – SIL constraints and safety contracts	86
10	Evaluation	88
10.1	Taking this work forwards	88
10.2	Critique	88
11	Conclusion	94
11.1	Overall Contribution to Safety Critical Systems Engineering	94
11.2	Dissertation Aim	94
11.3	Dissertation Approach	94
11.4	Dissertation Outcome	95
11.5	Costs and Benefits	95
11.6	Limitations	96
11.7	Further Work	96
12	References	98

Figures

Figure 1 Classes and objects	11
Figure 2 The Client/Supplier relationship.....	11
Figure 3 Classes and inheritance.....	12
Figure 4 Pre-conditions, post-conditions, invariants, and their inter-relationship	14
Figure 5 Generic modular system architecture	23
Figure 6 Possible communications and dependencies in a simple three-layer modular system	24
Figure 7 Three-unit modular system architectures	25
Figure 8 Three-unit modular system architectures and a potential interaction model	25
Figure 9 Reuse of safety evidence.....	27
Figure 10 Emergent properties as result of interactions.....	27
Figure 11 Safety case for a traditional system vs safety case for a modular system.....	28
Figure 12 IMA development lifecycle.....	30
Figure 13 Assumptions on modules within a modular system	31
Figure 14 The relationship between Design by Contract and Rely – Guarantee conditions.....	33
Figure 15 Satisfying pre-conditions and post-conditions	35
Figure 16 Deriving and consolidating safety requirements.....	36
Figure 17 Developing safety contracts.....	37
Figure 18 How software faults occur	40
Figure 19 White Box Testing vs Black Box Testing.....	40
Figure 20 Tests and regression tests	44
Figure 21 GSN pattern nomenclature	46
Figure 22 High-level system view of active steering system	52
Figure 23 Systems, sub-systems and functional units within an active steering system.....	52
Figure 24 Scenario 1 – A two module Client/Supplier system, with change in the Supplier.....	58
Figure 25 Validity of tests as regression tests based on pre-contract and post-contract changes	63
Figure 26 Process to identify regression tests	64
Figure 27 Regression test selection rules applied to weaker pre-contract/stronger post-contract	67
Figure 28 Regression test selection rules applied to weaker pre-contract/stronger post-contract	67
Figure 29 Generic Regression Testing.....	78
Figure 30 Generic identification of system DSRs and safety contracts	79
Figure 31 DSR and safety contract based regression testing (1 of 2, see also Figure 32).....	80
Figure 32 DSR and safety contract based regression testing (2 of 2, see also Figure 31).....	81
Figure 35 Scenario 2 – A two module Client/Supplier system with a timing restriction	83
Figure 36 Scenario 3 – A two module Client/Supplier system with a SIL restriction	85

1 Introduction

1.1 Background

Bespoke system development is perceived as risky. Projects have a habit of not delivering what they set out to deliver [4] [5]. Projects come in over time, and over budget, with de-scoping used as a means to deliver 'something'. The problem is particularly acute for bespoke systems including software within the system.

Modular systems are becoming more common [22], seen as a means of reducing risk by using a 'divide and conquer' strategy to split a single, large, and potentially complex system down into a collection of smaller, more manageable sub-systems, with functionality provided through modules. Where the modules are provided through 'commercial off the shelf' items, there is the advantage of being able to evaluate the finished module before purchase, so module quality, actual functionality, and any functional omissions are better understood. However, while the behaviour of individual modules may be well characterised, the joining together of modules develops emergent properties within a system. The emergent properties are introduced as a result of the interfaces that are necessary to join the modules together, and the interplay between messages passed between the modules, and the modules themselves.

Modular systems are dynamic in nature. Modules may be added, removed, or replaced. This may be done for a number of reasons, including system extension (e.g. to extend a system's functionality during an incremental development project), removal of functionality that is no longer required, and like-for-like change (e.g. for the correction of errors within the swapped module). As each updated system increment is created, the system should be tested, to ensure that new functionality is behaving in an acceptable manner, and pre-existing functionality has not regressed. Testing and regression testing need to consider both the properties of the modules within the system, and the emergent properties from the inter-working of the modules.

Where the modular system is safety critical or safety related, and one or more modules have been added, removed or updated, the safety case must also be updated to reflect the state of the updated system. Testing and regression testing will form part of the evidence to be presented within the updated safety case. The updated safety case must cover safety within the context of the whole system, covering both the individual modules, the interfaces, and the module interactions.

As Dijkstra observed [39]:

... program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence.

Testing (and therefore, regression testing) is difficult, and never 100% conclusive.

In her book [29], Leveson identifies a number of *Software Myths*. Some of these software myths can be extended to cover problems associated with modular systems. In particular:

Software Myth: Software is easy to change

Modular System Myth: Modular systems are easy to change

Changes to modules are easy (they are designed to be changeable). However, change always brings risk. New functionality within the system should be completely tested, and updated or untouched functionality within the system should be completely regression tested each time a change is made. Reacceptance is also required for safety critical and safety related systems.

Software Myth: Testing software ... can remove all the errors

Modular System Myth: Testing modular systems can remove all errors

Software is difficult to test. The large number of possible states within any significant system makes exhaustive testing impossible. Modular systems can also attain a large number of states. The inner working of the modules are often unknown (they are supplied as black boxes), and the system's emergent properties must be both identified and then tested (particularly in safety critical and safety related systems).

Software Myth: Reusing software increases safety

Modular System Myth: Reusing modules increases safety

Reuse may increase complacency. Modules may be developed for one application type but be used in other application types, or a system may be extended to cover functionality never initially envisaged within the system. In the safety domain, safety is a combination of the module and the environment within which it is used. The whole system needs to be tested, within the context of the emergent properties.

As a result, testing must be an integral part of modular system development. However, testing is time consuming and costly. Getting full coverage of the testable facets may be difficult. Investigating all possible executable routes through a system is even more difficult (or impossible).

Regression testing is also time consuming and costly. Where only small changes are made to a system, regression test costs can be out of all proportion with the size of the changes made. When a system is under constant change, or when a system is large, regression testing may be never ending. In addition to carrying out the actual testing activities, it is necessary to identify the regression test set. Selection of regression tests may be carried out in a number of ways, using systematic selection techniques or risk based heuristics. Identifying regression tests for a meaningful system can, in itself, place a heavy resource cost on a project. Where regression testing will be used to support safety case arguments, the selection of the regression test set must demonstrably deliver the true regression test set.

The object-oriented paradigm and Design by Contract may provide a rigorous solution to the problem of identifying regression tests for modular systems.

Within the object-oriented paradigm, classes can be thought of as units of functionality with interfaces. The inner workings of a class are hidden from the user. Instead, encapsulation is used to inform the user of the class' available functionality, routine inputs and routine outputs. Modules used within modular systems can be viewed in a similar way. The functionality and interfaces of a module are known, but the inner workings of a module are generally hidden from a systems integrator.

Design by Contract uses pre-conditions to specify the ranges of input values that are acceptable to a Supplier. Post-conditions are used to specify the range of output values, and the state of an object once a routine has terminated. Similarly, to be able to use modules effectively, it is necessary to understand the inputs that will ensure correct behaviour, and the outputs and states that will result from that correct behaviour

Design by Contract has been extended into the safety domain, through the development of safety contracts. Safety contracts make use of rely conditions and guarantee conditions to specify the required state of the system before a routine is called, and after a routine call has terminated.

The object-oriented paradigm makes use of inheritance, to transfer the properties of a higher level, more general class into a lower level, more specific class. Rules have been identified, relating the properties of the pre-conditions and post-conditions of the parent class to the properties of the pre-conditions and post-conditions of the child class. These rules are necessary to ensure that classes can be used interchangeably (i.e. so that it is possible to specify a parent class, but use a child class – since the child has inherited the properties of the parent, the child can provide the functionality of the parent). These rules also apply to the interchangeability of modules within modular systems, and can be used to identify regression tests.

1.2 Scope

This dissertation applies concepts from the object-oriented paradigm and Design by Contract to the problem of identifying regression tests for safety critical and safety related modular systems, where modules have been added, removed or updated within the system.

This dissertation identifies rules that can be used to select regression tests for integer functional safety contracts and integer non-functional safety contracts, a process for applying these selection rules, and safety cases patterns to support the updating of safety cases when following these rules.

1.3 Structure

Section 1 (this section) introduces the topics covered in this dissertation, and highlights the focus of the work contained in this dissertation.

Section 2 reviews literature pertinent to the dissertation, with particular focus on the object-oriented paradigm, Design by Contract, modular systems, testing, regression testing and the acceptance of safety critical systems.

Section 3 consolidates the literature survey, highlighting the problems associated with regression testing and the acceptance of modular systems.

Section 4 describes the system upon which examples within this dissertation are based, along with the working definitions, and initial assumptions.

Section 5 defines integer safety contracts, and the changes that may be made to integer safety contracts.

Section 6 considers the use of functional integer safety contracts as a route to identifying regression tests, where a Supplier has been updated within a Client/Supplier pairing.

Section 7 describes a process whereby integer safety contracts can be used to identify regression tests.

Section 8 presents two GSN patterns and a GSN module to support the process whereby integer safety contracts can be used to identify regression tests.

Section 9 considers the use of non-functional integer safety contracts as a route to identifying regression tests, where a Supplier has been updated within a Client/Supplier pairing.

Section 10 suggests how the work presented in this dissertation might be taken forwards, and provides an evaluation of the work to date.

Section 11 summarises the work presented in this dissertation, and suggests future work to extend the technique further.

1.4 Dissertation conventions

Bold text is used to highlight key words within the dissertation, and may be helpful to a reader trying to navigate this dissertation quickly.

Italic text is used to indicate text that is directly quoted from another source.

2 Literature Survey

This literature survey covers the subjects that underlie the work presented in the dissertation. The literature survey covers:

- Section 2.1 How the **object-oriented paradigm** may be used to develop interchangeable, reusable modular units of software based functionality, with **Design by Contract** being used to specify the contracts relating to the functionality.
- Section 2.2 How **Design by Contract** may be applied at a system level.
- Section 2.3 The architecture of **modular systems**, and the driver to modular systems.
- Section 2.4 Issues surrounding the safety **acceptance of modular systems** following initial development and incremental update.
- Section 2.5 How **emergent properties** can be described by extending **Design by Contract** to include rely conditions and guarantee conditions within **safety contracts**
- Section 2.6 How **contracts may be redefined during inheritance** within the object-oriented paradigm.
- Section 2.7 A **process for identifying safety contracts** within a layered modular architecture.
- Section 2.8 A **general introduction to testing**. Reasons for testing, and methods of testing are covered, along with the role of contracts and safety contracts within test specification.
- Section 2.9 A **general introduction to regression testing**, including identification of regression test sets.
- Section 2.10 The use of **Goal Structured Notation** to support **safety case** arguments.

2.1 Design by Contract

This section examines the object-oriented paradigm, focusing on software modularity and Design by Contract, as a means to developing reusable units of software.

2.1.1 What is a contract?

We live in a contractual world.

Some of the contracts that govern our lives are not formally recorded. These are informal contracts (i.e. they are verbally agreed, implicit or tacit contracts). For example, a shop will offer a Mars bar in exchange for valid money. There is no written contract. It is simply understood that as long as sufficient money is handed to the shop, a Mars bar will be given to the purchaser without any argument. It is also understood that if too much money is handed over by the purchaser to the shop, the purchaser will not only receive a Mars bar – the purchaser will receive both a Mars bar and some change.

Other contracts that govern our lives are formally recorded. For example, when a job offer is made, a contract of employment will normally be agreed between the employer and the prospective employee. A contract of employment may cover a number of details, depending on the nature of the job being offered. But, typically, a contract of employment will require the employee to provide their skill to the employer (i.e. work) for a certain number of hours in exchange for a sum of money being passed from the employer to the employee on a weekly or monthly basis.

In both cases, obligations are applied to the parties, and benefits are accrued by the parties:

	Obligation	Benefit
Shop	Gives Mars bar	Receives money
Purchaser	Gives money	Receives Mars bar
	Obligation	Benefit
Employer	Pays money	Can make use of employees time/skills
Employee	Makes time/skills available to employee	Receives money

A **formal contract** is used to record:

- the **obligations incurred** by the parties covered by the contract,
- the **benefits that may be accrued** by fulfilling these obligations, and
- the **penalties that may be incurred**, if the contract is broken.

2.1.2 The object-oriented paradigm

Design by Contract has its origins within object-oriented programming, and the development of the Eiffel programming language [1] [2] [3].

The object-oriented paradigm grew out of what was termed *the software crisis* [4] [5]. As computational power increased, so did the complexity of the programs written to harness this computational power. This made it increasingly difficult to write computer programs that were correct, understandable and verifiable. The software crisis manifested itself in a number of ways, including:

- projects running **late**,
- projects running **over budget**,
- software **quality being low**,
- software **not meeting user requirements**, and
- software code being **difficult to use, maintain and enhance**.

It is notable that while these issues were first recognised in the late 1960's, the issues are still very relevant today.

The object-oriented paradigm addressed the software crisis through a **strong emphasis on modularity** within software design and code. As the object-oriented paradigm evolved, different communities developed similar but different vocabularies to describe common key features. The terms used here are those used within the Eiffel community [3].

Object-oriented design has been described as:

... the construction of software systems as structured collections of abstract data type implementations, or "classes".

In particular:

- the **basic modular unit is the class**, used to describe one particular implementation of an abstract data type,
- a class is a model of an entity, seen from a particular view point – it is an **abstraction** of that entity,
- classes should be **designed as collections or clusters** of interesting and useful functionality, that may be reused in systems other than the system for which they were first designed,
- collections should **reflect the Client and inheritance relationships** that exist between classes,
- the emphasis is on **reusing whole data structures**, together with the associated operations (i.e. the reuse of whole classes), rather than on reusing isolated routines,
- **objects are instances of abstract data types**, know only via their official interfaces, and without knowledge of their internal workings,
- systems are structured around the **manipulation of objects, and the passing of messages** between objects (rather than on a flow of functions to be performed, as in functional programming).

As shown in Figure 1, a **class is a compile time notion**. A class is a description of an abstract data type (i.e. entity), written in such a way that it can be compiled. Conversely, **objects only ever exist at run time**. An object only ever exists in the computer's memory, or in data storage.

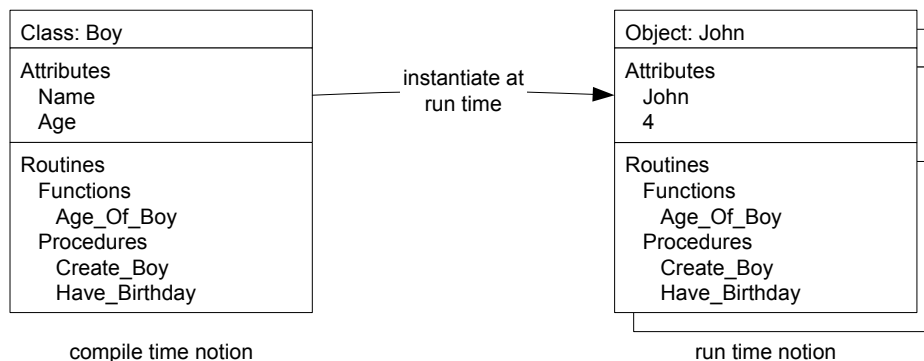


Figure 1 Classes and objects

A class is characterised by its **features**. These are the:

- **Attributes**, describing the class' structure via the class' variables.
- **Routines**, describing the manipulations that can be carried out on a class via the class' functions and procedures.
 - A **function** provides a means of querying the state of an object, by returning the value of an attribute, or by returning a value relating to the state of the object. Functions should not change the state of the object.
 - A **procedure** is a command, causing the state of the object to be changed. A procedure is not a query, and should not be used to return the value of an attribute, or any other value relating to the state of the object.

For example, the class `Boy` may:

- be described by two attributes, `Name` and `Age`,
- have one routine in the form of a function (`Age_Of_Boy`), which allows the age attribute of an object instantiated from the class `Boy` to be queried, and
- have two routines in the form of procedures (`Create_Boy`, and `Have_Birthday`), which allow an instance of `Boy` to be created as an object, and for the age of an object instantiated from the class `Boy` to be incremented by 1.

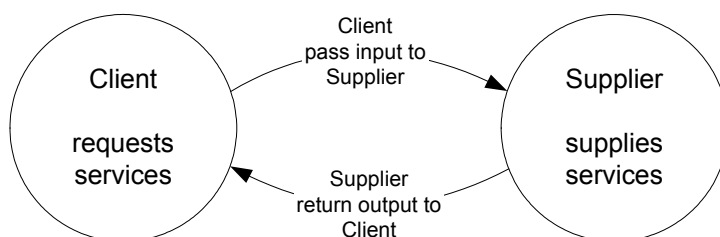


Figure 2 The Client/Supplier relationship

As shown in Figure 2, a class may call other classes. When a class calls another class, the calling class is a **Client**. The calling class expects to receive services from the called class – the called class becomes a **Supplier**. For example, a class `Register` may become a client of the class `Boy`, by making use of one or more entities of type `Boy`.

Objects are instances of a class. The class `Boy` describes all possible boys. The object `John` is one particular instance of the class `Boy`.

State is the set of values for the attributes of a particular object. For example, for an object of class `Boy`, created with the `Name` of `John`, and an `Age` of `4`, the state is (`John`, `4`).

Inheritance allows existing classes to be used as the starting point for new classes – one or more classes may be inherited into a new class, forming a parent/child relationship. An inheriting class (the child or

descendent class) has all the features (both attributes and routines) of the inherited (parent) class or classes. The inherited features may be redefined, while new features may be added, so creating more specialised versions of the inherited classes. Figure 3 shows a simple example of inheritance where the class `Boy` is inherited into a new class `Boy_Scout`. `Boy_Scout` has all the properties of `Boy` (e.g. Name, `Create_Boy`), but then goes on to extend `Boy` through the addition of new attributes and routines (e.g. `Patrol`, `Patrol_Belonged_To`, `Updated_Patrol`).

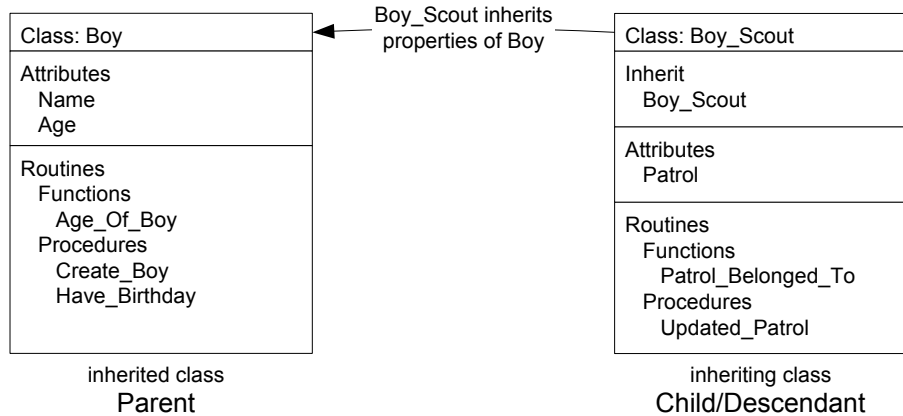


Figure 3 Classes and inheritance

One particularly important concept within the object-oriented paradigm is **encapsulation** [12]. Encapsulation is an extension of the concept of abstraction. A class is an abstraction of an entity, at both a structural and a behavioural level, where the internal working of that class are visible. Encapsulation is used to hide the inner working of a class. Encapsulation is used to describe a class' interfaces. To use a class, it is not necessary to understand the internal workings of that class. It is only necessary to know:

- which of an object's attributes can be read external to that object, and
- what routines are available to manipulate the attributes within an object, and so maintain the object's state.

Conversely, it is not necessary to know about attributes that are only accessible from and used within an object (i.e. attributes that are not accessible outside the object – this could include transient variables like loop controllers and variables used to allow sorting algorithms to function). Nor is it necessary to know about lower level routines that are called by a high level method, but are never called directly by a programmer accessing an object features. Encapsulation allows the programmer to know about the attributes and routines within the class that the programmer may access and use. Encapsulation hides all the other details, including how the routines are implemented. This is frequently described through the use of the word *public* to define the attributes and routines that may be called externally to the object, and *private* to hide the variable and routines that are never called externally to the object.

2.1.3 Reuse, integrity and reliability

As has been described, object-oriented programming leads to modular software. With modularity comes the opportunity for reuse. Reusable components become a commodity that can be maintained in collections or libraries, and used in a number of different situations. For example, the class `Boy` may be initially developed for use in a program to maintain a register of students within a school. But, the same class `Boy` might also be useable for a youth club membership listing, or as the starting point for a new class `Boy_Scout`. However, if components are to be reused, they must have reuse properties.

More generally, the components must be shown to be **reliable**. In this context, reliability is defined as [6]:

... a system's ability to perform its job according to the specification (correctness) and to handle abnormal situations (robustness). Put more simply, reliability is the absence of bugs.

Within safety critical systems, the components must also be of demonstrable **integrity**. In this context, integrity is defined as [54]:

Freedom from flaw or corruption.

Not only must a component behave according to its specification. The specification itself must also be correct. Otherwise, a component may be perfectly developed to a wholly imperfect specification.

If components are not seen to be reliable, with a correctness that can be trusted, they will not be reused. Instead of reuse, the 'not invented here' syndrome will prevail, and the wheel will be reinvented. However, developing reusable components is difficult. The components may be reused in many different applications [1], and in situations that were never originally envisaged. For example, a component originally developed for a low integrity application may be reused in a higher integrity application, where the potential consequences of incorrect behaviour could be more serious than was ever envisaged by the components original developers.

Within the object-oriented paradigm, there are a number of concepts that can help to make software more reliable. These include:

- **Static typing**, to help catch inconsistencies before they become bugs.
- **Garbage collection**, to help prevent errors caused by memory mismanagement.
- **Reuse component libraries**, produced and validated by reputable external sources, and (possibly) with additional validation having been carried out by earlier reuse. In effect, the component library becomes part of the standard set of development tools, alongside hardware, operating systems and compilers

Good understanding of the problem and problem domain is necessary to ensure correct specification. For reusable components this may be problematic, since it will be difficult for a component's developer to envisage all the uses to which a component may be put. Good documentation of a reusable component's properties is essential to counter this, and ensure reusability.

2.1.4 The fundamentals of Design by Contract

It has been argued that the object-oriented paradigm is not enough [6]. Instead, to be confident that object-oriented software components will operate reliably, it has been argued that a systematic approach is required to specifying and implementing object-oriented software components. To meet this requirement, the method of Design by Contract was proposed. Design by Contract has been described as [15]:

... an elegant synthesis of the essential concepts in three major fields of computing research: object-orientation, abstract data types and the theory of program verification and systematic program construction.

Within Design by Contract:

... a software system is viewed as a set of communicating components whose interaction is based on precisely defined specifications of the mutual obligations – contracts.

As described above, Design by Contract is based on work by Hoare [11] (among others), into the theory of program verification and systematic program construction. Hoare suggested that:

Computer programming is an exact science ...

where

... all the consequences of executing [a program] in any given environment can, in principle, be found out from the text of the program itself by means of purely deductive reasoning.

In other words, if we know the input to the program, and the logic of the program, it is possible to deduce the output from the program. This holds provided there are no emergent properties resulting from the running of the program (e.g. as in embedded systems, databases, and artificial intelligence).

Design by Contract applies the idea of formal contracts within the software domain, through the inclusion of assertions within object-oriented programs at the design and code stages of the software lifecycle. The links between the programming language Eiffel and Design by Contract are strong, to the point where it is difficult to see where one ends and the other begins. However, [7] attempts to separate Eiffel from Design by Contract, in a way that other publications (e.g. [1] [6] [13]) do not.

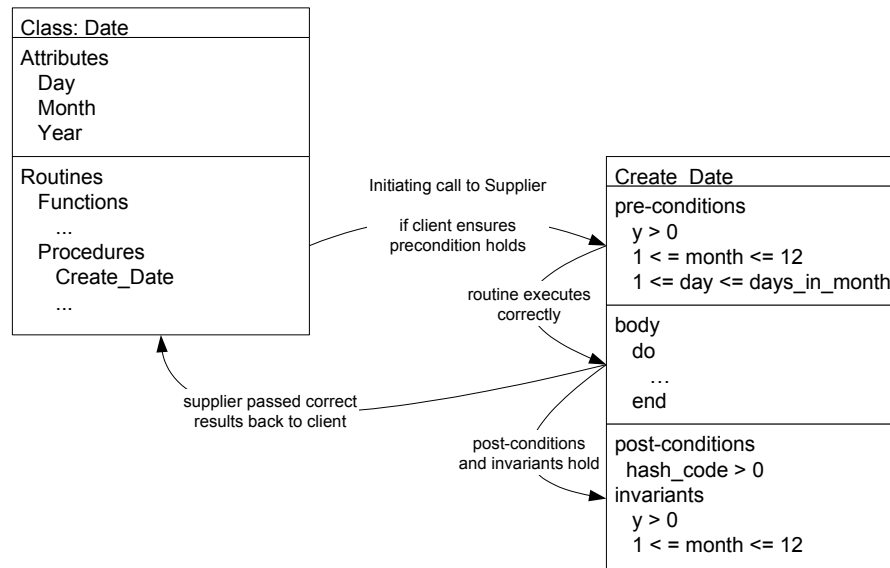


Figure 4 Pre-conditions, post-conditions, invariants, and their inter-relationship

Assertions are facts about a program that must be true for the program to execute in a reliable way. Assertions are Boolean statements (i.e. they can only ever be true or false). The chain of events is shown in Figure 4. There are three key assertion types:

- **Pre-conditions**, expressing the requirements that a Client must satisfy whenever it calls a Supplier routine. If a pre-condition does not hold (i.e. the pre-condition is false), the Client's call to the Supplier routine is invalid, and could potentially result in an unreliable outcome. At run-time, if a pre-condition does not hold, an exception would normally be raised. Pre-conditions describe both the initial state of the object, and the values passed to the Supplier routine.
- **Post-conditions**, expressing the conditions that the Supplier routine guarantees to return, provided that the pre-condition was satisfied on entry to the routine. Post-conditions only describe the final state of the object, differences between the initial and final state of the Supplier's object, and in the case of functions, the value of the returned value.
- **Invariants**, expressing conditions that must be satisfied by every instance of the class, whenever the instance is externally accessible (i.e. after the creation of an object, and after the completion of any call to a publicly available routine of the class). Invariants only describe the state of the object, when the object is stable, and not mid routine. Invariants are independent of the class' routines, and characterise the whole-life stable state [16].

Invariants only make sense within the object-oriented paradigm since they describe the stable state of an object. Pre-conditions and post-conditions can be used in more traditional approaches.

As Jacobson [13] describes it:

If you promise to call r with pre satisfied then I , in return, promise to deliver a final state in which $post$ is satisfied.

And, as Meyer [14] describes it:

First Law of software contracting:

There are only two ways a routine call may terminate: either the routine fulfils the contract, or it fails to fulfil it.

Second Law of software contracting:

If a routine fails to fulfil its contract, the current execution of its caller also fails to fulfil its own contract.

When a group of pre-conditions or post-conditions are listed, an implicit 'and' is assumed within the group. An implicit 'and' is also assumed between both the pre-conditions and post-conditions of any public routine.

For example, consider a class that allows a Client to create and maintain a date [8]. Dates within the Gregorian calendar conform to well-established rules (e.g. number of days in an Month, occurrence of leap years), which can be expressed via pre-conditions, post-condition and invariants.

- A date comprises of three attributes: day, month and year.
- To create a date, it is necessary to for a Client to supply three values, giving the value of the day, month and year of the date they wish to create.
- Pre-conditions to reliably and correctly creating a date might include:
 - The value of the year being greater than zero ($year > 0$)
 - The value of the month being between 1 and 12 ($1 \leq month \leq 12$)
 - The value of the day being consistent with the year and month chosen ($1 \leq day \leq days_in_month(input_month, input_year)$)
- Post-conditions to reliably and correctly creating a hash code for a created date might include:
 - The hash code not being negative ($hash_code > 0$)
- Invariants that will always apply to a created date might include:
 - The year always being greater than 0 ($year > 0$)
 - The month always being between 1 and 12 ($1 \leq month \leq 12$)

Assertions explicitly **tell the Client of a class the contract that is being entered into** [3]. The pre-conditions bind the Client. The post-conditions bind the Supplier. Invariants state what must always be so at the end of any routine (i.e. when the object is in a stable state, and not currently executing a routine).

2.1.5 The principles of Design by Contract – writing a 'good' contract

In addition to providing a separation between Eiffel and Design by Contract, [7] [10] also make suggestions on the principles that underlie the principles of Design by Contract. These include:

- 1 **Separate queries from commands.** A query should only be made via a function, and should only ever read back values about the state of an object. A function should not change the state of the object. Functions are used to provide runtime checkability. Conversely, a command should only be made via a procedure. While a procedure may change the state of an object, it should not be used to read back values about the state of an object.
- 2 **Separate basic queries from derived queries.** Derived queries should not 'just happen'. Basic queries provide the building blocks from which to construct derived queries. As a result, once the basic queries are understood, the derived queries should also be understood.

- 3 **For each derived query, write a post-condition** that specifies what result will be returned in terms of one or more basic queries. By writing a post-condition within a derived query in terms of one or more basic queries, the post-condition is describing the contract explicitly, and showing the assumptions upon which the derived query is based.
- 4 **For each command, write a post-condition** that specifies the value of every basic query [changed by the command]. By writing a post-condition within a command in terms of one or more basic queries, the post-condition is describing the contract explicitly, and showing the visible changes of state that happen when the command is executed.
- 5 **For every query and command, decide on a suitable pre-condition.** Pre-conditions tell the Client the terms of the contract under which the functions and procedures can be expected to execute reliably.
- 6 **Write invariants to define unchanged properties of objects.** Invariants should be used to help the user build a complete model of the abstraction contained within the class. To this end, invariants should include those that can be easily derived from the pre-conditions and post-conditions, as well as that require a constructed argument.

In all cases, the aim is to inform and educate the user of the class about the nature of the contract into which they are expected to enter when using the class.

2.1.6 The benefits of Design by Contract

The benefits of Design by Contract are claimed to include [2] [3] [6] [7] [9]:

- **Better design**
 Assertions describe the assumptions that the developer made when writing the code. The development of assertions encourages the programmer to think more deeply about the code under development, and what they are trying to achieve from a number of different angles. This helps to lead to designs that are more systematic, clearer and simpler than would normally be achieved. Additionally, the breaching of the contracts embedded in assertions provides a consistent exception raising mechanism.
- **Improved reliability**
 Better design of the code through a more conscious development process leads to greater understanding of what is being developed, and so greater reliability. The assertions can be consistently checked at run time, allowing easy testing to check that the contracts embedded within the assertions are being fulfilled. Where an exception is raised as a result of assertions being violated, the exception can be handled in a consistent manner.
- **Better documentation**
 The assertions form part of the public, encapsulated view of a class, describing a class in terms of the observable behaviour (but not in terms of their internal implementation), and in terms of the contract that is in place during usage. The
 - routine names and attribute profiles,
 - routine assertions and class invariants, coupled with
 - initial comments provided within a routine to describe its purpose
 describe the class' encapsulation, and should (if correctly done) provide the programmer with all the information they need to successfully use/reuse the classes. Using defined syntax and tools allows documentation to be automatically derived from the developed artefacts.
- **Inbuilt test mechanism**
 The assertions also serve as a specification for the class, and can act as a test oracle that indicates the expected results for a given input. Where a test case satisfies the pre-conditions, the output must satisfy the post-conditions, and the invariants. If the output does not satisfy the post-condition and

invariants, an exception will be raised – there may be no need to write an explicit test to check the outcome from running a routine.

- **Easier debugging**

Assertions describe the limits of expected behaviour.

- Pre-conditions provide information about what is going on within the program by detecting interface irregularities between object calls whenever they happen, and at whatever level of test is being undertaken.
- Post-conditions check the validity of a single class, showing how methods respond to legal input, by checking the output from every legal invocation of a routine.

Using assertions allows for the precise point of contract breakdown to be identified.

- **Support for code reuse**

With good documentation comes the possibility of reuse – the user is told what the classes can achieve, and under what circumstances the correct results will be achieved. Runtime checking of assertions provides the user with additional support.

- **The removal of defensive programming**

Defensive programming can be used to protect a program against a routine being called with inputs outside the specific inputs for which the routine was written, or when the routine call is inappropriate to the state of the system. With defensive programming it is necessary to do through more extensive coding what can be achieved quickly and succinctly, in a highly visible manner, through the use of assertions within the context of Design by Contract.

2.1.7 The costs and limits of Design by Contract

However, there are downsides to Design by Contract [7]. In particular:

- **It takes time to write useful contracts.** The pressure to appear productive early in the software cycle may mean that lines of code win out over quality of code, and the development of meaningful contracts.
- **It takes practice to write useful contracts.** Skilled programmers may not be available, making it necessary to
 - provide training, and
 - accept reduced productivity while programmers gain the necessary experience.
- Contracts can improve programs, but **do not make perfect programs**, since.
 - contracts cannot express all the desirable properties of programs, and
 - there is always some scope to make errors.
- Where speed of program development is the primary concern, **slowing development** to increase quality may not be acceptable (but, of course, more haste makes less speed).
- Contracts work **best for sequential programming**, where pre-conditions can be checked prior execution, rather than checking that the correct action has been taken after execution.
- There is **limited language support for contracts**. Eiffel provides support for contracts within the language. Other languages rely on tools to provide support for contracts.

2.1.8 An extension to Design by Contract

Pre-conditions and post-conditions are concerned with the input to an object and the output from an object. They are not concerned with the state of the object. However, the concept of pre-conditions and post-conditions can be extended to the state of the object [38] via:

- **accepting conditions**, used at the same time as pre-conditions, to ensure that an incoming message is acceptable to the object's state, as the calling message is received, and
- **resulting conditions**, used at the same time as post-conditions, to ensure that the resultant object state is valid, given the accepting state and the procedure called.

2.1.9 Conclusion

The object-oriented paradigm provides a framework for developing software as interchangeable units of functionality. Design by Contract extends this by providing a means of enforcing contractually binding interface specifications between Clients and Suppliers, through the use of pre-conditions, post-conditions and invariants. One of the side effects of Design by Contract is that the contracts can be used to support test development by providing a framework for testing, by specifying the ranges of inputs and outputs.

2.2 Design by Contract at the System Level

As covered in Section 2.1, Design by Contract is grounded in object-oriented programming, and the Eiffel programming language, using assertions to define contracts at the class level. However, there are a number of higher-level system modelling methods/notations that also make use of the ideas presented within Design by Contract. These include the less well known:

- Analysis by Contract,

as well as better known methods/notations:

- Unified Modelling Language and the Object Constraint Language,
- SysML, and the
- Business Object Notation.

This section examines these means of supporting Design by Contract at the system level.

2.2.1 Analysis by Contract

Design by Contract is covered in detail in [7]. However, the book then goes on to consider how contracts can be used in analysis models, at a level higher than program design.

A process can be modelled by a use case [7]. The use case tracks a sequence of interactions between systems and/or people within a process. Typically, the interactions are constrained by rules, which can be defined in terms of contracts or assertions.

For example, consider a process that allows a bank's Client to withdraw money from their bank account.

- Pre-conditions to reliably and correctly make a cash withdrawal might include:
 - the bank's Client holding an account with the bank, and
 - the balance on the account being greater than the amount of money that is to be taken from the account.
- Post-conditions following a cash withdrawal might include:
 - the bank's Client has the requested amount of cash in their hand, and
 - the account balance is reduced by the amount of the cash withdrawal.
- Invariants that will always apply might include:
 - the balance on the account is always greater than 0.

It is not clear from [7] how Analysis by Contract is to be integrated into the system development environment. However, one possible approach is given in [24], where this type of approach is taken to eliciting contracts relating to derived safety requirements.

2.2.2 Unified Modeling Language and Object Constraint Language

The Unified Modeling Language (UML) is a modelling notation, with its roots in object-oriented methods [17][18][19]. UML is closely associated with modelling software intensive systems at the specification, design, and build levels. However, UML is much more versatile than this, and has found use within a far wider range of application areas, including higher-level systems design and business process analysis.

UML is based on a number of concepts and constructs, which can be used to model a system from different viewpoints and at different levels of abstraction. Starting at the world level, UML can be used to define the system boundary, and describe how the system of interest will interact with entities outside the system boundary. At the other end of the scale, UML can be used to model message passing at the object level.

The models fall into two main areas:

- Structural Models

The structural models describe entities within the system, and their relationships to other entities. These views provide the basics, upon which the dynamic behaviour can be based. The structural views do not model time dependent behaviour. The structural models are based on the:

- class diagram,
- use case diagram,
- component diagram, and
- deployment diagram.

- Dynamic Models

The dynamic models describe the behaviour of the system over time. The life history of entities can be described, and communication patterns modelled. The dynamic models are based on the:

- statechart diagram,
- activity diagram,
- sequence diagram, and
- collaboration diagram.

UML offers support for the use of pre-conditions, post-conditions and invariants.

In UML, a **pre-condition** is defined as [18]:

A constraint that must be true when an operation is called.

A **post-condition** is defined as:

A constraint that must be true at the completion of an operation.

Invariants are not defined in [18]. However, [17] defines an **invariant** as:

A constraint that must be true at all times (or, at least, at all times when no operation is incomplete).

Pre-conditions should not be confused with guard conditions. A **guard condition** is described [18] as:

A condition that must be satisfied in order to enable an associated transition to fire.

A guard condition is used to shield an operation; it is an 'if' statement. A guard condition is a query, returning a Boolean value, but does not change the system or its states. Where the guard condition returns 'true', the system then moves on to the operation, where the pre-condition expresses the requirements that the Client must satisfy (i.e. the contract) for the operation to execute reliably.

The UML books and standard do not specify the way in which pre-conditions, post-conditions and invariants should be written; natural language is acceptable. However, natural language is inexact, and often results in ambiguity. A number of formal languages have been developed to overcome this problem. However, to be understood, formal languages generally need a strong mathematical background, typically not found among business and system modellers. The Object Constraint Language (OCL) [20][21] was developed as a formal language (so allowing rigorous, unambiguous descriptions), based on fixed syntax text rather than mathematical symbols (so allowing a wider user base). OCL can be used for a number of different purposes, including specification of:

- Invariants on classes and types in a class model. For example, in a company where the number of employees must always be greater than 50, this can be specified as:

context Company **inv:**
 NumberOfEmployees > 50

- Pre-conditions and post-conditions on routines. For example, a company may keep details of an employee's spouse. In a function that returns the name of a spouse, there could be a pre-condition that the employee is married. This can be specified as:

context Employee::getCurrentSpouse() : Person
pre: Employee.isMarried = true

2.2.3 SysML

SysML is based on UML [47]. However, SysML is aimed at the systems engineering level, so is smaller than UML, and no longer contains the software centric diagrams. SysML takes seven of the UML diagrams, and includes a further two diagrams. The models fall into three main areas:

- Structural Models

The structural models describe entities within the system, and their relationships to other entities. These views provide the basics, upon which the dynamic behaviour can be based. The structural views do not model time dependent behaviour. The structural models are based on the:

- block diagram, c.f. the UML class diagram,
- internal block diagram, c.f. the UML component diagram,
- parametric diagrams, for performing quantitative and qualitative analysis and
- package diagram, c.f. the UML deployment diagram.

- Dynamic Models

The dynamic models describe the behaviour of the system over time. The life history of entities can be described, and communication patterns modelled. The dynamic models are based on the:

- statechart diagram,
- activity diagram (modified from UML 2),
- sequence diagram, and
- collaboration diagram, c.f. the UML use case diagram.

- Requirements Model

Classified as both structural and dynamic. The requirements model is based on the:

- requirements diagram, supported with satisfaction arguments used to demonstrate how requirements are met by design elements.

Support for pre-conditions, post-conditions and invariants remains.

2.2.4 Business Object Notation

Business Object notation (BON) is a method for the analysis and design of object-oriented systems [15]. The emphasis is on:

- **Seamlessness**, so allowing the design to flow from the modelling environment, into the coding environment, without having to go through a step change in format.
- **Reversibility**, so allowing modifications to be maintained between the analysis, the design and the implementation, in both a forwards and backwards direction.
- **Software contracting**, so providing support for Design by Contract.

BON does not offer the wide range of structural and dynamic models offered within UML, since these models are believed to be too far removed from the coding environment in which the model must ultimately be implemented to offer a seamless and reversible interface. Instead, BON relies on object-oriented modelling concepts, modelling the system as:

- a **set of classes**, via the system chart, cluster chart, and class chart (the static model), related by
- **inheritance** and **Client dependencies**, based on
- **strong typing** and **software contracts**, using.
- the **event** chart, **scenario** chart, and **creation** chart (the dynamic model).

BON offers support for the use of pre-conditions, post-conditions and invariants.

In BON, a **pre-condition** [15]

... states a predicate that must be true when the feature is called by a client.

A **post-condition**

... states a predicate that must be true when the feature has been executed and the supplier object returns control to the client.

An **invariant**

... must always be true both before a visible feature is called by a client and after feature execution, just before returning to the client.

BON allows for use of a textual form (closer to natural language) and a graphical form (closer to formal language).

2.2.5 Conclusion

The object-oriented paradigm and Design by Contract can be used to develop modular software, with well defined interface specifications. However, the same design principles and notations can be used at a system level to specify modules, and the interfaces between modules. A number of methods and notations are available to support the principles of Design by Contract at the system level (e.g. UML, the object constrain language, and the Business Object Notation).

2.3 Modular Systems

Section 2.1 presented the support available for the development of reusable code modules, through the use of the object-oriented paradigm, and Design by Contract. Section 2.2 considered how Design by Contract may be supported at the system level.

This section examines the commercial trend towards modular systems, where a number of components are bought (possibly from a number of different suppliers) and/or developed separately, with development work being based on the integration of these components. This section also looks at the nature of modules, and the drivers leading to the development of modular systems. An overview of a typical modular system is presented.

2.3.1 Modules

Modular systems are made up of modules or components. A module can be thought of as an entity that delivers functionality, via its **inner workings**. Depending on the source, the component may be white box (i.e. the implementer fully understands the inner working), gray box (i.e. the implementer knows something about the inner working), or black box (i.e. implementer knows nothing about the inner workings).

A component will also:

- **Have an interface**, to allow communication with the rest of the system.
- **Interact** with other components within the system.
- Be **usable within one or more contexts**. At one end of the spectrum, a component may be very specialised, and exist as a bespoke item, for use in a single, specific system. At the other end of the spectrum, the component may be very general with many possible uses (e.g. Linux, or the Microsoft operating system).

As such, components have many the properties associated with the object-oriented paradigm.

2.3.2 The trend to modularity

Some of the trends leading to an increasingly modular approach to systems development are noted in [22]. These trends include:

- Increasing **complexity within a single system** (e.g. operating systems, and the trend to bloatware), with a corresponding increase in components and lines of code. Modularity gives a 'divide and conquer' approach to developing complex systems.
- Increasing system complexity with designed for and **intentional integration between systems** (e.g. car stabilising systems, where the steering system, the braking system and the suspension system are three independent items that must work as an integrated unit), with a corresponding increase in the number of systems to be integrated.
- Increasing system complexity with **opportunistic integration between systems** (e.g. air traffic control, where existing systems are being integrated with new systems), with a corresponding increase in the number of systems to be integrated.
- A **move to distributed systems**, often partially based on commercially available components (e.g. integrated modular avionics).
- **Pressure to use Commercial Off the Shelf (COTS) products**, for cost management and risk mitigation. COTS products invariably have to be integrated with additional bespoke software, and possibly integrated with other COTS products, or bespoke developments.

The advantages of modularity include:

- Being able to develop a **component that can be used in a number of similar applications**. And, conversely, being able to have immediate access to a suitable module, without having to go through the development lifecycle.
- Being able to **develop applications in a stepwise manner**, adding modules and so extending functionality over a period of time.
- Being able to **replace modules**, as better alternatives become available, or as the needs of the application change.

However, there are also problems to be overcome:

- **Safety is a property of the system as a whole.** Any collection of modules working together as a system has to be demonstrated as safe within the context of the entire system (and not just the individual modules).
- **Changing a single module within the system requires regression testing** of the whole system. This can be costly in terms of both time and resource, and may make modular systems less attractive overall.

2.3.3 Anatomy of a modular system

In generic terms, a modular system can be thought of as having three layers, as shown in Figure 5 [23] [24][32]:

- The **hardware layer** is taken to contain the physical components of the computer system, including monitors, keyboards, circuit boards, processors and memory chips. This layer may also connect with other systems via associated hardware (e.g. Ethernet). However, hardware can also be extended to include actuators and mechanical items driven by actuators.
- The **middle layer** contains the operating system. The operating system provides the interface between applications and hardware, managing hardware resources and processing application commands. Operating systems may themselves have applications to manage the operating system.
- The **application layer** provides the reason for the systems existence. The application layer may contain one or more applications, each offering desired facets of functionality.

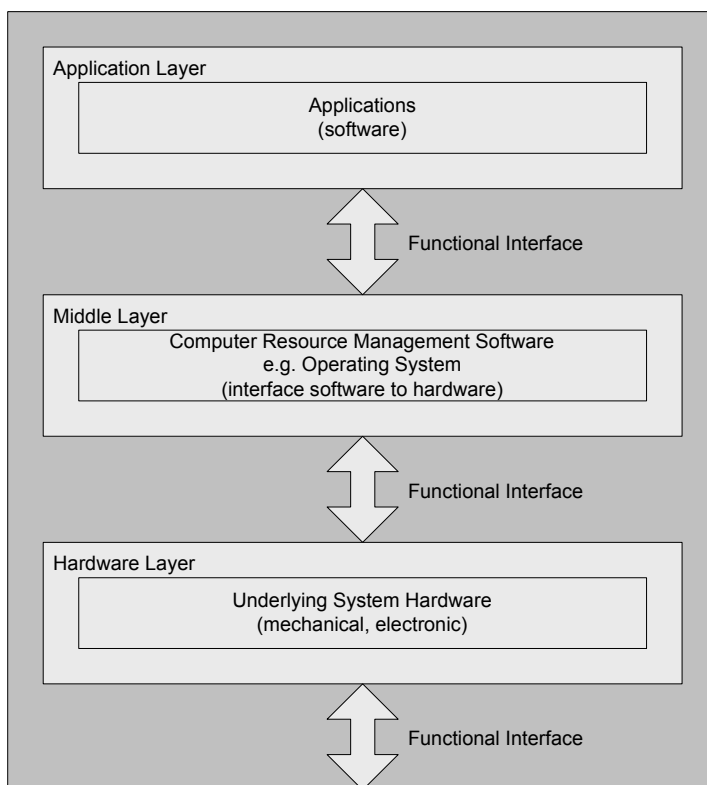


Figure 5 Generic modular system architecture

The middle layer and hardware layer are the most likely to be comprised of commercially available products. The application layer may contain commercially available products. However, since this is the most customised part of the system (to support a specific application), there is a more general trend to bespoke modules.

In **traditional systems**, the systems are:

- designed and built as a whole, with a single configuration, for a specific purpose, and
- built from specific bespoke components (possibly with some off the shelf components), that are not expected to change over a system's lifetime.

Modular systems are the opposite of this. Modular systems are:

- designed and built around available components, with bespoke development being carried out only under specific conditions (e.g. to integrate components, to provide functionality not available off the shelf),
- based on common, publicly defined interfaces, that allow easy interchange of modules,
- 'black box' constructions, where the inner working of a significant part of the system is not transparent to the integrators and testers,
- often developed incrementally, with the core infrastructure and core functionality being put in place (and possibly released for use), before the addition of further functionality,
- built with the expectation that components will be exchanged over a system's lifetime, as updated components become available (e.g. to improve performance, to extend functionality), and
- potentially flexible enough to change module configuration on a use by use basis.

Traditional systems tend to be fixed and constant. Modular systems are designed and built to change.

There is also an additional, hybrid, category. A traditional system may have been in use for some time, but be reaching the limit of its usefulness in its current form (e.g. electronic components are obsolescent and cannot be replaced or repaired due to a lack of suitable components). Rather than develop a new system to replace the existing system, it may be desirable to extend the life of the existing system by replacing modules within that system. In this case a traditional system morphs into a modular system.

2.3.4 Interfaces in modular systems – Operational system level

Figure 5 and Figure 11 show 'classic' three-layer modular system architectures. A worst-case implication for internal system communications and dependencies in a relatively simple three-layer modular system is shown in Figure 6.

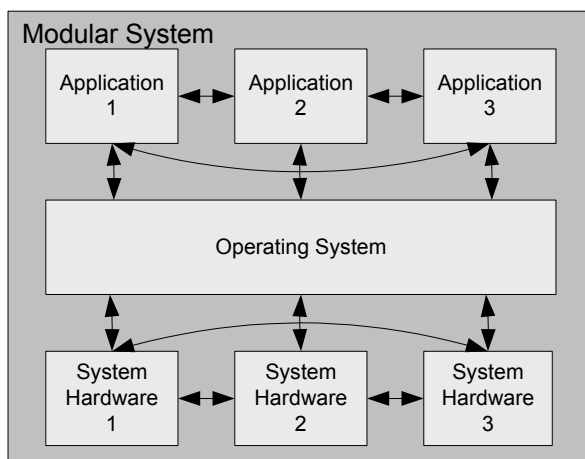


Figure 6 Possible communications and dependencies in a simple three-layer modular system

There are additional complications. Although the operating system is supposed to allow the applications and system hardware to be independent of each other, it can be argued [24] that:

- since the operating functionality depends on the system hardware, the applications are also dependent on the system hardware, and
- there may be direct links between the applications and the system hardware,

an alternative three-unit modular system architecture model should be considered, as shown in Figure 7, where the system hardware directly underpins both the applications and the operating system.

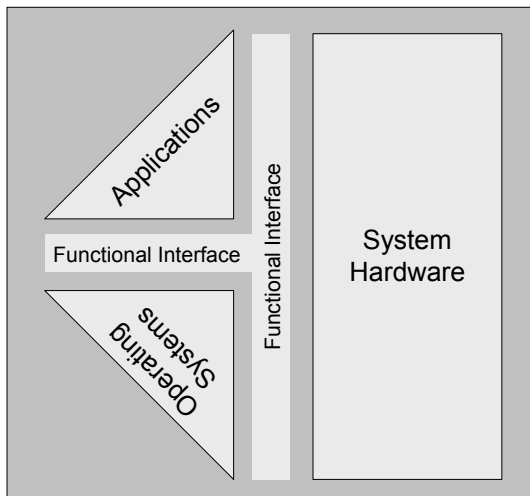


Figure 7 Three-unit modular system architectures

This can again lead to a large number of interactions, as demonstrated by the simple model shown in Figure 8.

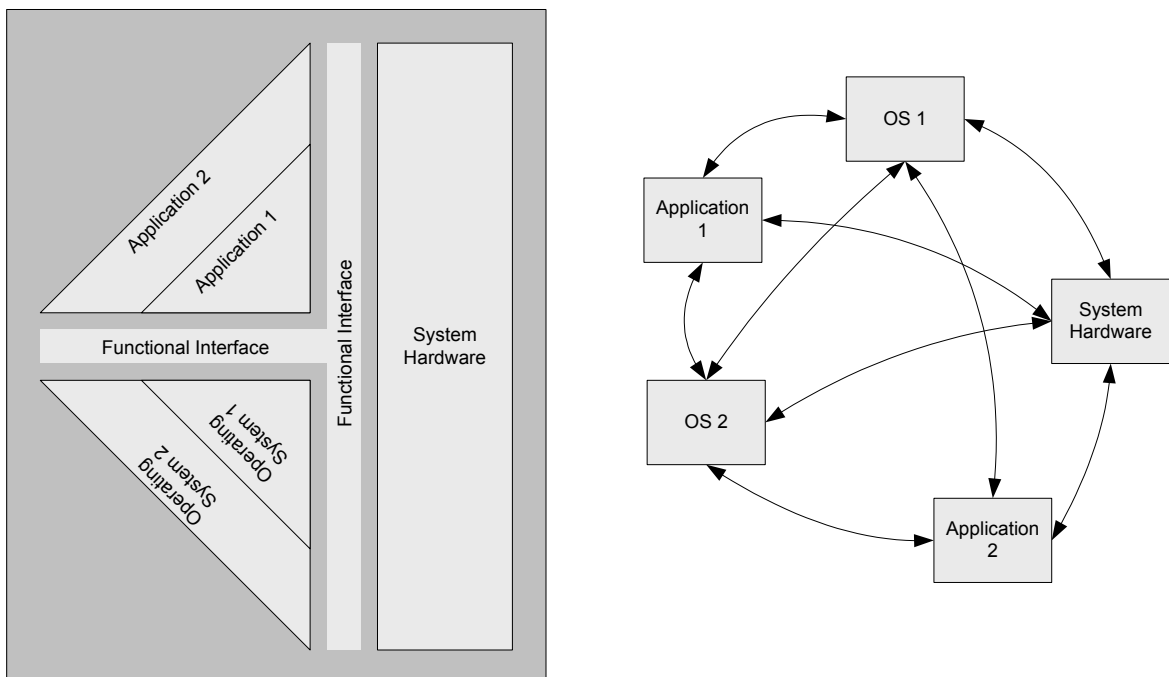


Figure 8 Three-unit modular system architectures and a potential interaction model

2.3.5 Conclusion

Modular systems are based on reusable components, with interfaces that allow the components to interact with other components within a system. As a result of the integration interfaces, the integration of modules also results in emergent properties. So, while the individual modules used within a system can be tested and assured as standalone units, it is also necessary to test and assure the system as a whole.

2.4 Acceptance and incremental acceptance of modular systems

In common with other safety critical and safety related systems, modular systems in the safety domain must be accepted (and possibly certified) for their intended use. In this Section, the facets of safety acceptance are considered. The potential for the reuse of safety evidence is examined. A process for the incremental acceptance of an integrated modular avionics system is presented.

2.4.1 Acceptance of modular systems

At a point, or a number of points within a development lifecycle, the item under development has to make a transition, and become an item available for use. The release of a developed item is generally accompanied by acceptance (and possibly certification) activities. As part of the acceptance activities, an authorised entity will need to gain sufficient evidence of the system's:

- **accuracy** (i.e. the design specification has been correctly implemented), and
- **functionality** (i.e. the requirements specification is satisfied),

such that there is enough confidence that the system will function as expected in the operating environment [25][26]. Acceptance may occur in a single step, incrementally, or in phases.

Acceptance activities will generally include final testing of the item under development. However, acceptance generally also requires that the associated documentation have been delivered and that the development environment was fit for purpose. Additionally, for a safety critical system, it will be necessary to show that:

- the deliverables and the development environment **comply with one or more standards**, and
- the system functions with acceptable integrity under all possible scenarios, including being **acceptably safe even when things go wrong**.

Increasingly, acceptance is achieved through the development of a safety case, where a safety case can be described as [27]:

A structured argument, supported by a body of evidence that provides a compelling, comprehensible and valid case that a system is safe for a given application in a given operating environment.

Safety cases need to address a number of areas, including [23]:

- **Design safety:** is the design acceptably safe in all its operational states?
- **Maintenance safety:** what maintenance is necessary to ensure continued safe operation?
- **Operational safety:** how must the system be operated to ensure safe operation?
- **Personal health and safety:** are there operator health and safety issues that must be addressed?
- **Environmental safety:** are there environmental issues associated with the systems construction and operation?
- **Disposal safety:** what are the health and safety issues that must be addressed during disposal?

Safety cases can be developed to suit a number of different development types, including [26]:

- **Generic product** safety case, presenting evidence that a generic product is acceptably safe within a variety of applications,
- **Generic application** safety case, presenting evidence that a generic product is acceptably safe within a specific class of applications, and
- **Specific application** safety case, presenting evidence that a specific product is acceptably safe within one specific application,

and development stages, including:

- **Limited operation** safety case, presenting evidence that limited operation of an additional system alongside a pre-existing system is acceptably safe within a specific situation, and
- **Full operation** safety case, presenting evidence that operation of a system is acceptably safe within a specific situation.

One aim of these different safety cases is to allow effective re-use of safety evidence. As shown in Figure 9, the safety evidence goes from being general (relating to a product or application) to specific (relating to a generic product being used for a specific application within a specific product). The safety case for the specific application is able to refer to the safety case for the generic application and generic product.

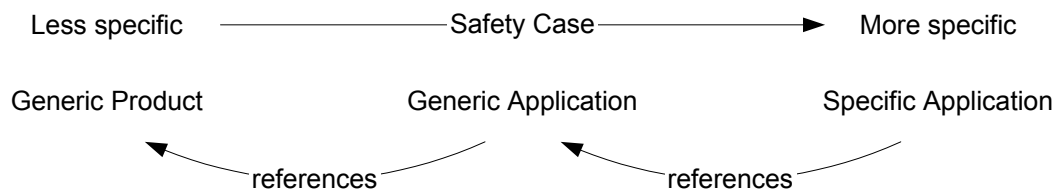


Figure 9 Reuse of safety evidence

However, this is simplistic. As [29] observes:

System safety deals with systems as a whole rather than with subsystems or components.

Safety is an emergent property of systems, not a component property. One of the principle responsibilities of system safety is to evaluate the interfaces between the system components and determine the effects of component interaction ...

Emergent properties occur in hierarchical structures. As lower level components are combined to form a higher level component, new properties emerge as a result of the interactions between the combined components (see Figure 10). The safety of a system is dependent on the contribution from both the:

- safety of the individual sub-systems, and the
- safety of the interactions between the sub-systems.

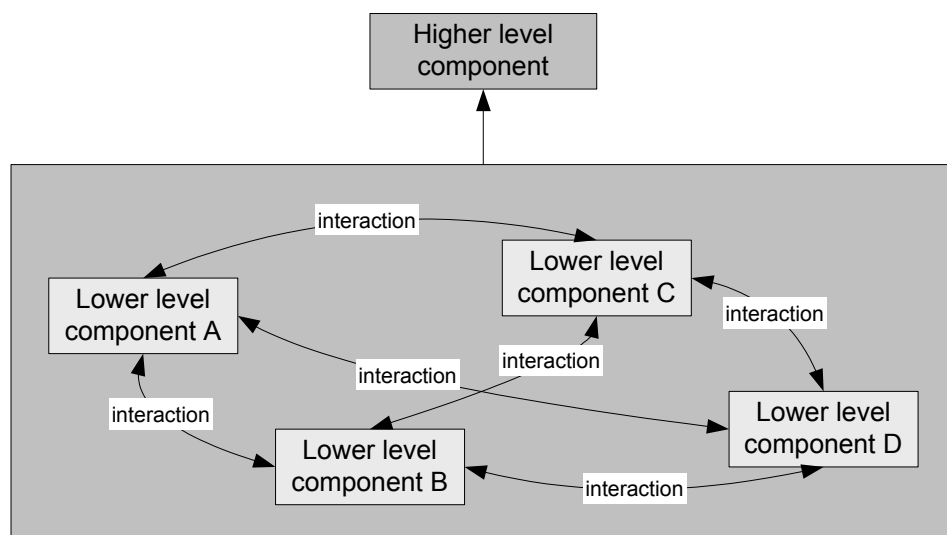


Figure 10 Emergent properties as result of interactions

As a result of the emergent properties, it is not sufficient to show that each of the sub-systems is acceptably safe. It is also necessary to show that the system as a whole (including the interactions) is acceptable safe [30][31] [32]. Safety cases need to consider the interactions between the sub-systems and not just the sub-systems themselves, since:

- a failure in one part of the system may show itself in another part of the system, as a result of a flow of actions and interactions across the system, and
- a failure detected in one component may need to be managed by another component.

As has been discussed in Section 2.3.3, traditionally systems have been built as an immutable single unit, while modular systems comprising a number of sub-systems are built around interchangeable units that can be updated, removed and added to over the lifetime of the system. Figure 11 shows the effect this has on the safety evidence and safety cases of these two classifications of systems:

- For the traditional system, a single set of safety evidence and a single safety case are required, at the time the system first goes into service. Since the system will remain fundamentally unchanged over its lifetime, little if any update will be required at a later date. Internal interfaces between the elements within the system do not need to be considered as items in their own right. As a result, when one part of the system is changed, the effect across the whole system must be evaluated, since the whole is tightly bound. – this is costly and time consuming.
- For the modular system, each of the constituent modules is a bounded, standalone, entity. Each module may have associated safety evidence, and possibly a generic safety case. However, the safety evidence and any associated safety case will only cover the functionality within the module boundary. The individual modules also need unifying safety evidence and a unifying safety case to cover the emergent properties, and so demonstrate that the system as a whole (i.e. the interacting modules and any integration applications) is acceptably safe for its intended purpose. As a result, the interfaces must be considered within the safety analysis [30].

Since a modular system is designed with change in mind, the modules within the system may change over the lifetime of the system. This will cause the emergent properties to change. New interactions may be added, while other interactions may be removed. As a result, new failure modes may emerge, and existing failure modes may change. Sources of common cause failure may be incorporated into the system [33].

This means that the existing system level safety case has to be revisited and updated as necessary to demonstrate that the updated system as a whole (i.e. the interacting modules and any integration applications) continues to be acceptably safe for its intended purpose.

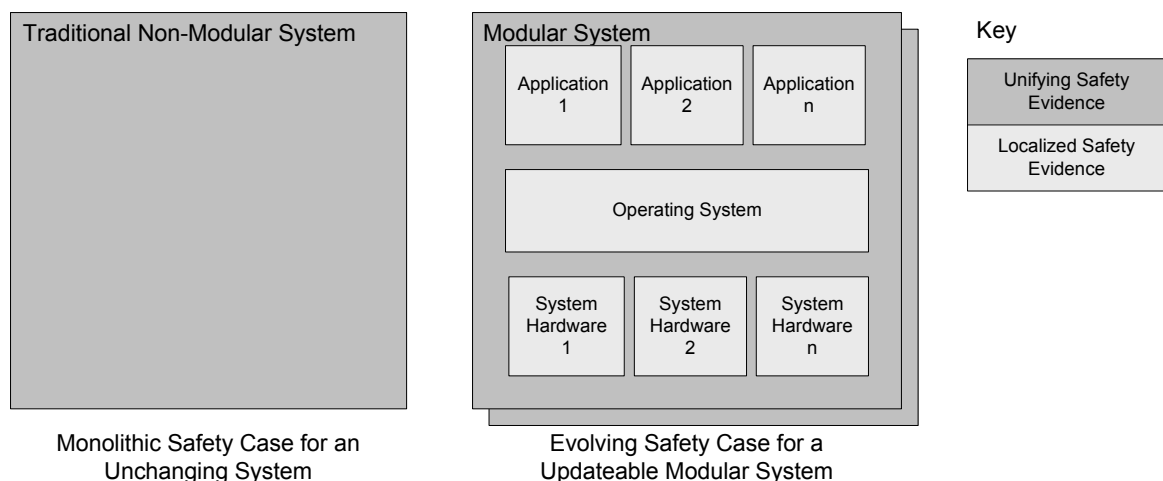


Figure 11 Safety case for a traditional system vs safety case for a modular system

2.4.2 Incremental acceptance of modular systems

Incremental acceptance is evolving in response to the demand for incremental development, where different parts of a system are developed independently, over different timescales, with a view to 'plug and play' type functionality. After initial acceptance of a system developed from interchangeable modules, the system may be changed. Possible reasons for change include:

- extending the functionality through the addition of one or more new modules and associated interfaces,
- removing part of the functionality either by switching off parts of one or more modules, or by removing one or more modules and their associated interfaces, or
- changes to one or more modules (e.g. as a result of a newer module being made available, or as a result of applying a patch).

After each incremental change, the system must be re-accepted, to cover the incremental change. This is termed incremental acceptance.

Integrated modular avionics are at the leading edge of incremental acceptance. Integrated modular avionics are described as [46]:

... is a shared set of flexible, reusable, and interoperable hardware and software resources that, when integrated, form a platform that provides services, designed and verified to a defined set of safety and performance requirements, to host applications performing aircraft functions.

Integrated modular avionics are based on:

- an aircraft wide network, linking
- modular control units, where
- each modular system has defined tasks to carry out, but can call on data from other modular systems as input, and to allow systems to work together as an integrated whole,

and are motivated by:

- functional integration for improved services,
- reducing the number of boxes and cabling to save weight, and
- the need to make hardware transparent, so managing obsolescence and supporting upgradeability.

Integrated modular avionics are consistent with the three-layer architecture in Figure 5. They have been developed as a means of [24] allowing aerospace applications to be ported to new hardware with minimal code changes, through the use of standardised operating systems, so allow applications to be moved between aircraft with little change, and at reduced cost. Similar systems are being developed in other sectors, including the automotive industry.

The RTCA and EUROCAE WG60/SC200 committee has developed guidance and certification considerations for integrated modular avionics. Within the context of integrated modular avionics, the WG60/SC200 committee have described incremental acceptance as [46]:

A process for obtaining credit toward approval and certification by accepting or finding that an IMA module, application, and/or off-aircraft IMA system complies with specific requirements. This incremental acceptance is divided into tasks. Credit granted for individual tasks contributes to the overall certification goal. Incremental acceptance provides the ability to integrate and accept new applications and/or modules in an IMA system, and maintain existing applications and/or modules, without the need for reacceptance.

As shown in Figure 12, the working group identified six tasks:

- **Task 1: Module acceptance**, where modules with identifiable, standalone functionality, are developed and accepted for a specific operating system.
- **Task 2: Application software/hardware acceptance**, where the accepted modules are integrated and accepted within a specific hardware environment.
- **Task 3: IMA system acceptance**, where accepted applications are brought together and accepted as an integrated system.
- **Task 4: Aircraft integration of IMA system**, where an accepted system is integrated and accepted within a particular aircraft.
- **Task 5: Change** of modules or applications.
- **Task 6: Reuse** of modules or applications.

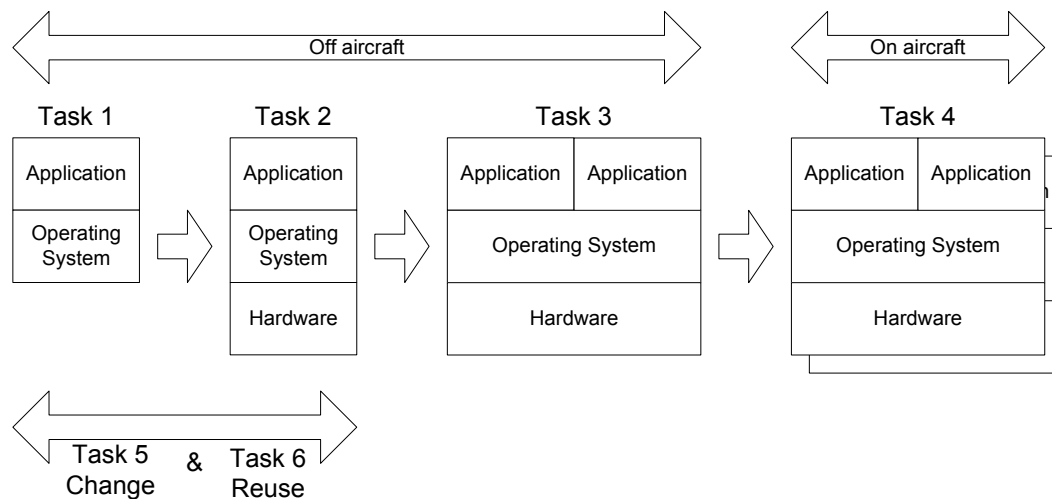


Figure 12 IMA development lifecycle

The emphasis is on accruing acceptance evidence (including test evidence) incrementally, with a focus on being able to transfer acceptance evidence from earlier tasks to later tasks, and between usage situations. Effectively, this means producing localised safety evidence for the individual modules within the system, but then producing unifying safety evidence for the whole system.

2.4.3 Conclusion

Acceptance of modules can occur at a number of levels (e.g. generic product, generic application, specific application), with the more general acceptance building towards the case of the more specific acceptance. Acceptance of modular systems may also be carried out incrementally, with stepwise changes in functionality, as modules are added, removed or updated.

2.5 Emergent properties and Design by Contract

As discussed in Section 2.4, safety is an emergent property of the system as a whole. It is not enough to show that each of the sub-systems is acceptably safe. It is also necessary to show that the interactions between the sub-systems are acceptably safe. To do this, it is necessary to identify and understand the nature of the interactions between the sub-systems. This section covers the identification of sub-system interactions.

2.5.1 Rely – Guarantee reasoning

The use of a modular system architecture leads to emergent properties. Not only does the safety of the modules have to be demonstrated. It is also necessary to demonstrate that the modules interact in an

acceptably safe way – we need to be able to understand and argue about the safety of the emergent properties.

The problem is described in [30]. Given a modular system containing two modules, as shown in Figure 13, how is it possible to show that Module 1 is acceptably safe for operation in the Modular System, without some knowledge of Module 2 (and visa versa)?

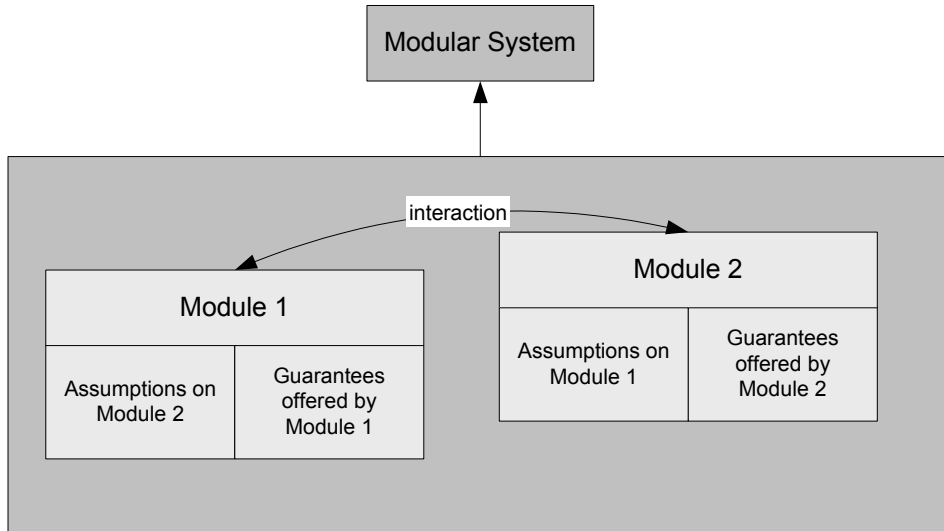


Figure 13 Assumptions on modules within a modular system

[30] proposes the use of **assumptions** and **guarantees**. In the Modular System shown in Figure 13, it is possible to define the assumptions placed on Module 2, when considering the operation of Module 1 (and visa versa). Once the assumptions have been defined, it is then possible to demonstrate that:

- the assumptions hold, or
- the assumptions are incorrect, and take appropriate action.

This type of argument is used in computer science for validation and verification (but not acceptance and certification) activities, where it is called **assume – guarantee reasoning** [35][36]. Assume – guarantee reasoning can be viewed as:

... a “divide-and-conquer” approach that infers global system properties by checking individual components in isolation.

where

... a [a module] M guarantees a property P when it is part of a system that satisfies an assumption A ...

While verification checks that a system works correctly, certification for safety related systems also has to ensure that a system acceptably manages and limits the consequences of a module (or group of modules) malfunction. As a result, the assumptions placed on Module 2 when considering the operation of Module 1 must include assumptions about the way Module 2 behaves when Module 2 has failed (and visa versa). In some cases, it may be possible to establish that a module will function in a safe way even when there are no assumptions on any other modules. A lack of assumptions does not mean that there are no interactions. Where there are no assumptions, the module without assumptions must be able to function correctly regardless of how any of the other modules within the system are functioning and/or malfunctioning.

Since a safety related system needs to be able to operate under both normal and abnormal conditions, it is necessary to identify assumptions for both normal and abnormal operational conditions. It is also necessary to ensure that there are no knock-on effects, with failures propagating in an uncontrolled

manner. For example, if Module 1 fails and the Module 1 guarantees degrade, it is likely that the Module 2 guarantees will also degrade. It would be undesirable for the Module 1 guarantees to then further degrade, so causing the Module 2 guarantees to then further degrade (i.e. for a negative feedback loop to be established), since it is likely that the end point would be no remaining, meaningful, guarantees being associated with the degraded forms of Module 1 and Module 2.

When using assume – guarantee, it is implicit that the failure of Module 1 can only impact on Module 2 through their respective assumptions and guarantees. The assumptions and guarantees will concern the values and relationships between various shared states and variables. However, for assume – guarantee reasoning to work, it is necessary to expect that **partitioning** is enforced. Partitioning is the mechanism by which multiple applications running within the same system are kept separate, and each given access to their own independent resources. With partitioning, the interface between Module 1 and Module 2 will remain even after one of these modules has failed so that private variables and state are not affected. As a result of partitioning, the module that has not failed will continue to have the ability to access Module 3, Module 4 ... Module n, and perform its functions.

Within safety related literature [22][23][32][34], assume – guarantee reasoning tends to be referred to as **rely – guarantee conditions**. However, the principles remain unaltered.

2.5.2 The relationship between Design by Contract and Rely – Guarantee conditions

At first glance, there is little difference between pre-conditions/post-conditions and rely/guarantee conditions. However, closer examination reveals some significant differences relating to the location of the conditions. These differences are demonstrated in Figure 14.

Pre-conditions and post-conditions relate to the relationship between a Client (requesting a service) and a Supplier (the entity that will supply the service). Traditionally, the Client and the Supplier are defined within two software code objects. However, the Client and the Supplier could be two modules within a modular system, where one module requests a service from another module.

- The pre-condition must hold when the Client initiates a service with the Supplier (otherwise the execution outcome cannot be guaranteed, and mayhem may ensue).
Pre-conditions are between the Client and the Supplier, with the focus being on the Client. They describe the service call state, but not the system state.
- The post-condition describes the state of the system after the service has been provided.
Post-conditions are between the Supplier and the system, with the focus being on the system. They describe the system state after execution has terminated.

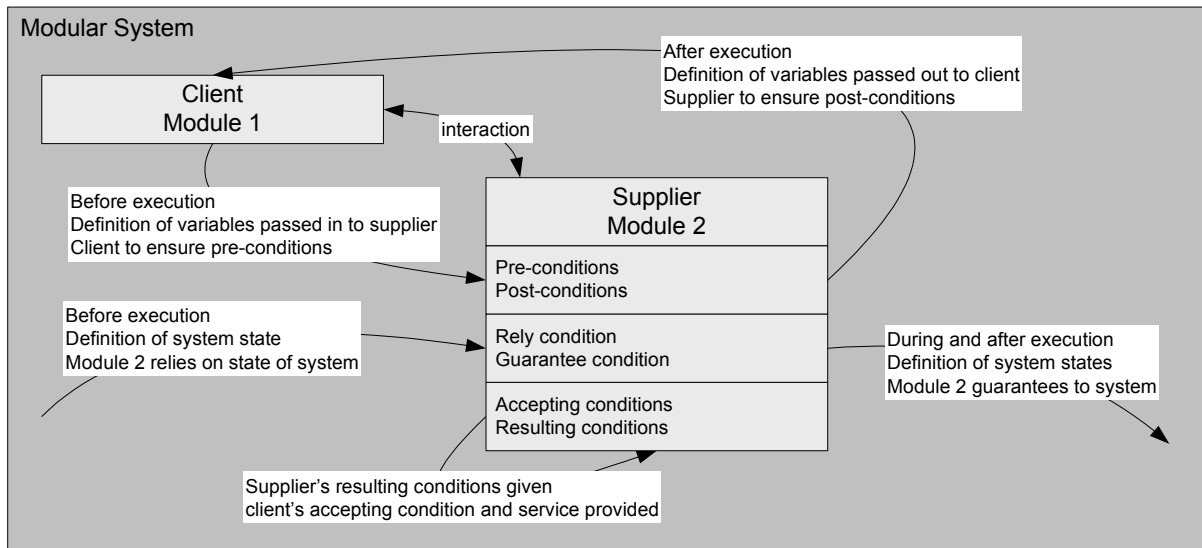


Figure 14 The relationship between Design by Contract and Rely – Guarantee conditions

Rely and guarantee conditions relate to relationships between the modules within a modular system.

- Rely conditions describe the system wide assumptions that have been made before execution to justify the safety argument.
Rely conditions describe system state before execution.
- **Guarantee conditions describe the system state** after execution.

Accepting and resulting conditions work with pre-conditions to cover the state of the Supplier.

- Accepting conditions couple the incoming message from the Client with the Supplier state, to ensure that the Supplier is able to supply the requested service given its own internal state and the content of the incoming message from the Client.
Accepting conditions are between the Client and the Supplier, with the focus being on the Supplier's state.
- Resulting conditions describe the Supplier's state after execution, in the light of the Suppliers state before execution, and the nature of the service supplied.
Resulting conditions describe the Supplier state after execution.

2.5.3 Conclusion

Within Design by Contract, pre-conditions and post-conditions specify the contract between a Client and a Supplier. Rely conditions and guarantee conditions specify the contract between the Supplier and the rest of the system before and after a function has been executed. Safety contracts are specified by the combined use of pre-conditions, post-conditions, rely conditions and guarantee conditions.

2.6 Redefining and matching contracts

The object-oriented paradigm focuses on the development of units of functionality. Encapsulation is used to hide the inner workings of the code. Instead, only the attributes and routines available to the user of the code are made public. The object-oriented paradigm also makes use of inheritance, where a more general unit of functionality can be inherited into a more specialised unit of functionality, to suit a more tightly scoped situation.

Design by Contract extends this by providing a mechanism for defining the pre-condition and post-condition that forms a contract between a Supplier and its Clients. Rules apply to how pre-conditions and post-conditions may vary when inheritance is invoked. This section considers these rules in more detail.

2.6.1 Contracts and inheritance

As discussed in Section 2.1.2, Design by Contract has its origins within object-oriented programming. One of the concepts within object-oriented programming is the use of inheritance. When inheritance is used, all the properties of the parent class or parent classes are inherited into the child class. Inheritance includes all the assertions, with the parent assertions being inherited by the inheriting child class.

Rules for redefining the inherited assertions within a child class are given in a number of places, including [7]. It is possible to:

*... weaken the [inherited] preconditions.
... strengthen the [inherited] postconditions.*

or

*... add more clauses to the [inherited] invariant, which are and-ed onto the inherited clauses,
thus strengthening the inherited invariant.*

where:

- a weaker contract is a less restrictive contract, and
- a stronger contract is a more restrictive contract,

without breaking the ability to specify a parent type, but then substitute a child type (one of the purposes of inheritance, since a child is a more specialised version of its parent), and still meet the pre-conditions and post-conditions.

While these rules have been defined for object-oriented programming, the rules also have relevance within Design by Contract. As show in Figure 15:

- If a contract is to be upheld, the Client must satisfy the pre-conditions of the Supplier. The Supplier's pre-conditions may be exactly satisfied, or the offering from the Client may be contained within the Supplier's pre-conditions. If a pre-condition is only partially satisfied, or a pre-condition is not satisfied, then (depending on the runtime setup) an error may be trapped and managed, or an unpredictable outcome may occur.
- Likewise, the Client must be satisfied with what the Supplier returns. As long as the Client's expectations sit within the Supplier's post-condition, all is well. Where the Client's expectations are only partially satisfied, or are totally unsatisfied, the Client may not be able to gracefully managed the returned values.

As a result, if a Supplier is updated, the pre-condition must remain as it was, or become weaker, if there are to be no knock-on effects to unchanged Clients. Likewise, the broader a pre-condition, the more likely a Client (any Client) is likely to be able to satisfy the pre-condition.

Conversely, if a Supplier is updated, the post-condition must remain as it was, or become stronger, if there are to be no knock-on effects to unchanged Clients. Likewise, the narrower a post-condition, the more likely a Client (any Client) is to be satisfied by the returned values.

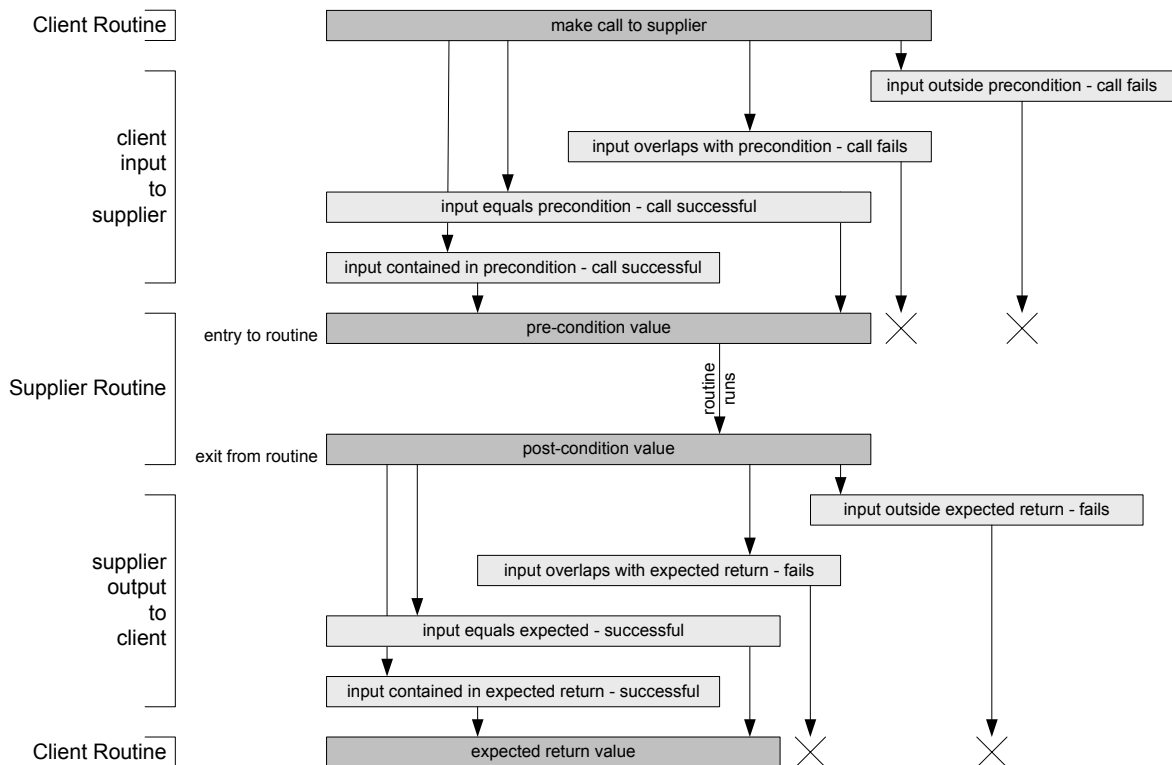


Figure 15 Satisfying pre-conditions and post-conditions

2.6.2 Conclusion

Within the object-oriented paradigm, inheritance is used as a means of specifying high level, generic classes where the properties can then be inherited into lower level, more specific classes, to provide more tailored services. By specifying a high level generic class, it is possible to interchange the high level generic class with any class that inherits from that class. Within inheritance, pre-conditions may be weakened, and post-conditions may be strengthened without removing the ability to use inheritance. The same rules can be to the interchange of modules within a modular system.

2.7 Finding safety contracts in layered architectures

A two-phase process for finding contracts in layered, modular architectures (with particular emphasis on safety contracts, where safety contracts are a fusion of pre-conditions, post-conditions, rely conditions and guarantee conditions) is given in [24][32][48]. . This section describes this process.

2.7.1 Phase 1

As shown in Figure 16, the **first phase** aims to consolidate the safety requirements, and identify the safety dependencies. It also defined the constraints on component reuse by identifying the relationships between modules.

- Phase 1, Step 1: carry out **system analysis** to:
 - identify system level hazards,
 - produce an architecture structure diagram, and
 - summarise the role of each component in the architecture.
- Phase 1, Step 2: for each component in the architecture, identify possible **component failures** and then:
 - analyse how each of these failures could impact system safety, and
 - derive appropriate control and mitigation strategies, and the associated derived safety requirements.

- Phase 1, Step 3: **consolidate the derived safety requirements**, to ensure consistency. The derived safety requirements will fall into one of three types:
 - Type 1 DSRs: the requirements on a component addressing failures internal to that component (e.g. the requirements placed on Component 1, as a result of a failure in Component 1),
 - Type 2 DSRs: the requirements on a component addressing failures external to that component (e.g. the requirements placed on Component 1 as a result of a failure in Component 2), and
 - Type 3 DSRs: the requirements on a component resulting from the analysis of another component (e.g. the requirements placed on Component n as a result of the analysis on Component 1).

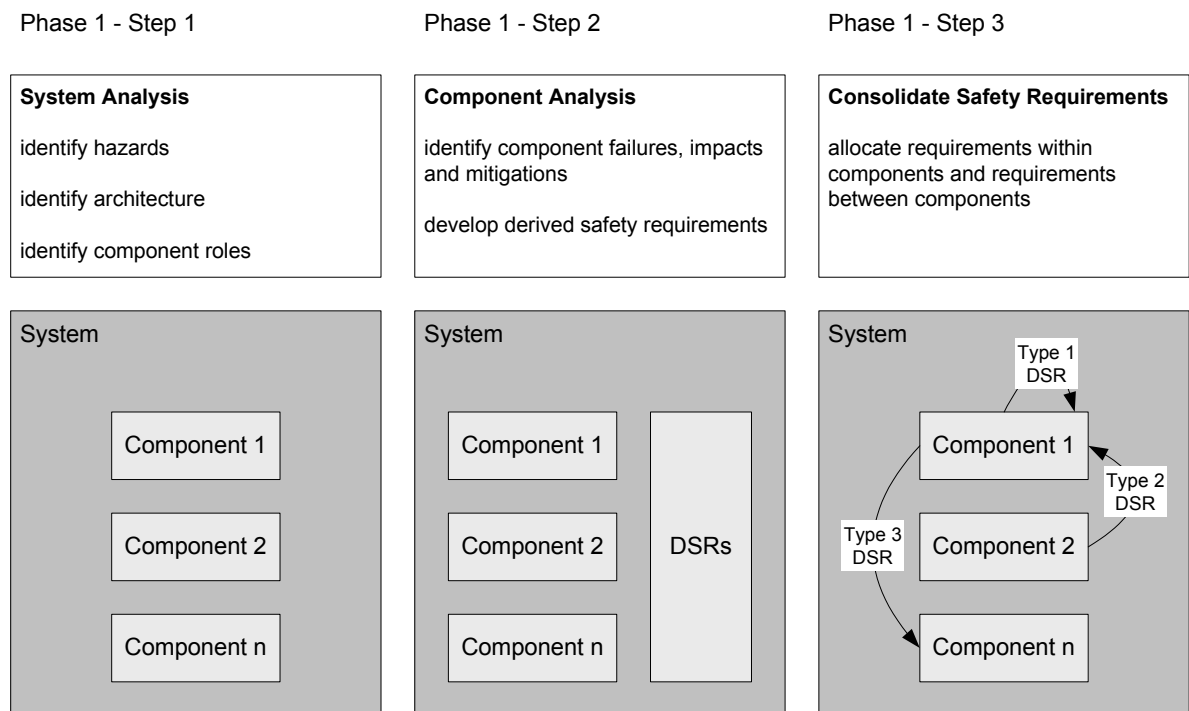


Figure 16 Deriving and consolidating safety requirements

2.7.2 Phase 2

A safety contract is based on:

- the pre-condition and rely condition, defining pre-operation states as required by the Supplier, and
- the post-condition and guarantee condition, defining post-operation states delivered to the Client and system by the Supplier,

and

- may be met by more than one function within a component, and
- may rely on software outside the control of the Client.

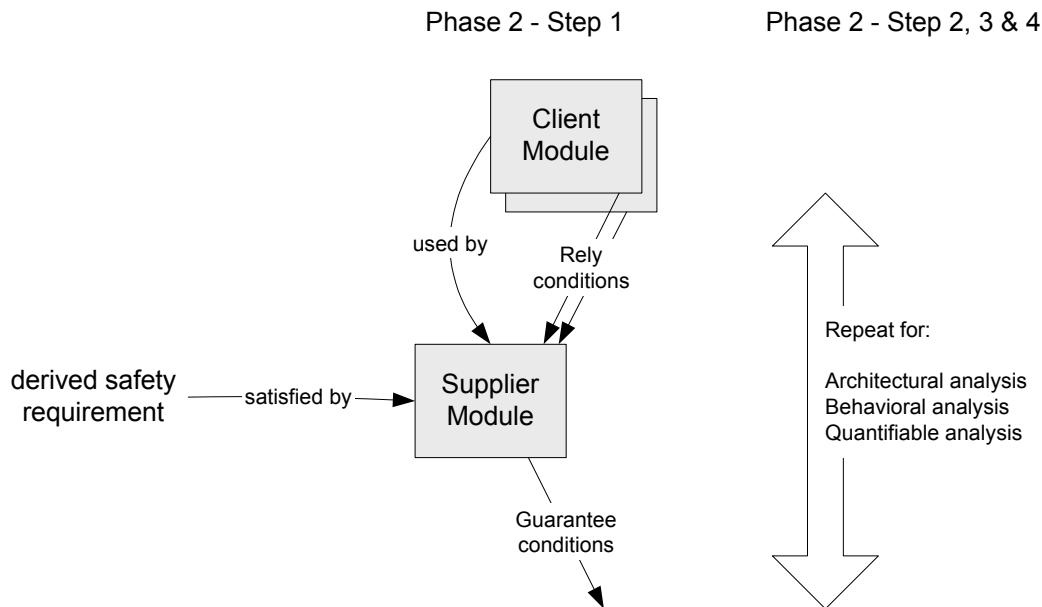


Figure 17 Developing safety contracts

As shown in Figure 17, the **second phase** develops safety contracts between the components, to support the consolidated derived safety requirements. It also defined the constraints on component substitution and upgrade. Four levels of abstraction have been identified.

- 1 Phase 2, Step 1: for each derived safety requirement, **identify the Supplier module** that will supply the service, **and the Clients** that will make use of the services.
- 2 Phase 2, Step 2: for each Supplier module, identify the **architectural decisions** (e.g. hardware choices, component structures, hardware/component relationships) that impact on the derived safety requirements. Define the **rely conditions and the guarantee conditions that the architecture places** on each of the Supplier modules.
- 3 Phase 2, Step 3: for each Supplier module, identify the **behaviours** (i.e. the way in which components behave) that impact on the derived safety requirements. Define the **rely conditions and the guarantee conditions that the behaviours place** on each of the Supplier modules.
- 4 Phase 2, Step 4: for each Supplier module, identify the **quantifiable requirements** (e.g. syntactic, performance, failure rate, and reliability requirements) that impact on the derived safety requirements. Define the **rely conditions and the guarantee conditions that the quantifiable requirements place** on each of the Supplier modules.

2.7.3 Conclusion

Safety contract can be developed in a two-phase process. Firstly, the system hazards are identified, and ascribed to component failures, to develop derived safety requirements. Secondly, for each derived safety requirement, the Client and Supplier modules are identified. The architectural decisions, component behaviours and quantifiable requirements that impact on the derived safety requirements are identified, along with the rely conditions and guarantee conditions that the architectural decisions, component behaviours and quantifiable requirements place on each Supplier module.

2.8 Testing

Section 2.7 describes a means of identifying safety contracts within a modular, layered architecture. In addition to documenting the behaviour of units of functionality, contracts may be used as a means of specifying the tests used to ensure that a system meets its requirements.

This section discusses testing in general. It then goes on to consider the use of contracts as a means to identify and develop system tests.

2.8.1 What is testing?

As a system is developed, there is a need to ensure that each phase of the development has been performed correctly, and that the original specifications for the development were correct. These are the processes of verification and validation, where [37]:

***Verification** is the process of determining whether the output of a lifecycle phase fulfils the requirements specified in the previous phase.*

and so demonstrate that:

... the output of a lifecycle phase conforms to the input, rather than show that the output is actually correct ...

while

***Validation** is the process of confirming that the specification of a phase, or of the complete system, is appropriate and is consistent with the customer requirements.*

and

***Testing** is the process used to verify or validate a system or its components.*

Tests fail when the item under test is unable to satisfy a specified requirement within specified limits. Within code, failure occurs as the result of missing or incorrect code, and is always **systematic**. Systematic failures are not random. Rather, systematic failures occur because of a fault in the specification or development process (e.g. incorrect specification, incorrect code). When the correct set of circumstances occurs, a systematic fault will always manifest itself.

Testing generally forms part of the acceptance process. Testing demonstrates correctness of functionality. The process under which testing is carried out, and the results that are obtained may be used to demonstrate conformance to standards and/or legislation.

2.8.2 Levels of testing

Testing may be carried out at a number of points in a project.

Within object-oriented software development, testing typically includes [38]:

- **Unit testing** on a relatively small executable (e.g. a single object of a class, or a cluster containing a few related classes).
- **Integration testing** on a complete system or coherent subsystem of software and hardware software units.
- **System testing** of a complete, integrated application.

More typically, testing is carried out in stages that follow the steps within a project's lifecycle. Typically, testing includes [49]:

- **Component testing**
 - **Unit testing** on an individual component, to ensure that it operates correctly.
 - **Module testing** on a collection of dependent components.
- **Integration testing**

- **Sub-system testing** on a collection of modules that have been integrated into a sub-system. Since a sub-system may be independently designed, for bringing together with other sub-systems from other sources at a later point in time, testing for interface errors is particularly important at this stage.
- **System testing** on a collection of sub-systems integrated to form an entire system. The focus is on testing that there are no unexpected and unwanted interactions between the sub-systems within the system, and validating that the system meets its functional and non-functional requirements.
- **User testing or acceptance testing** where users or another accepting body check that the system is fit for purpose under operational conditions. Errors and omissions in system requirements are often identified at this stage. Performance may also be identified as an issue.

In each case, testing is carried out against a baseline specification that describes the behaviour and properties required of the system at that point in the lifecycle. At lower levels of testing (e.g. unit testing), the focus is on functionality. As a project moves to higher levels of integration and more meaningful units of functionality, the focus in testing is increasingly towards testing interfaces and emergent properties.

2.8.3 Types of testing

Typically, testing is split into functional and non functional testing. Qualities that may be specified and then tested against include [38] [50]:

- **Functional testing**
 - **Functionality**, including suitability, accuracy, interoperability, compliance and security.
 - **Error management**, to ensure the correct implementation of error mitigation and recovery routines.
- **Non-functional testing**
 - **Reliability**, including maturity, recoverability, fault tolerance, volume testing and stress testing.
 - **Usability**, focusing on the user interfaces, and including learnability, understandability, operability and the usefulness of help information.
 - **Efficiency**, including time behaviour and resource behaviour.
 - **Maintainability**, including stability, analysability, changeability and testability.
 - **Portability**, including installability, replaceability and adaptability.

While functional testing is focused on inputs and outputs, non-functional testing covers a wide range of qualities.

Interestingly, safety testing is not identified in this list, or in similar lists from other sources. However, by analogy, safety testing relates to ensuring that the safety requirements identified as being necessary to prevent or mitigate failures that impact on system safety have been acceptably implemented. Safety requirements may cover a whole range of issues, and result in safety requirements necessitating one or more of the types of testing identified above.

2.8.4 Why test?

Currently, humans produce software. As a result, any useful software will contain faults, where [37]:

A fault is a defect within the system.

If a fault is exercised in a suitable way, it will manifest itself as a failure, where [37]

A system failure occurs when the system fails to perform its required function.

Possible failures include incorrect output, unexpected termination of the system, or a violation of any of the other requirements placed on the system. As Figure 18 shows, faults occur for a number of reasons [37][38]:

- A requirement may be missed, with no implementation being made to satisfy the requirement.
- Additional functionality, not specified in the requirements, may be added.
- An attempt may be made to satisfy a requirement, but the implementation is incorrect.

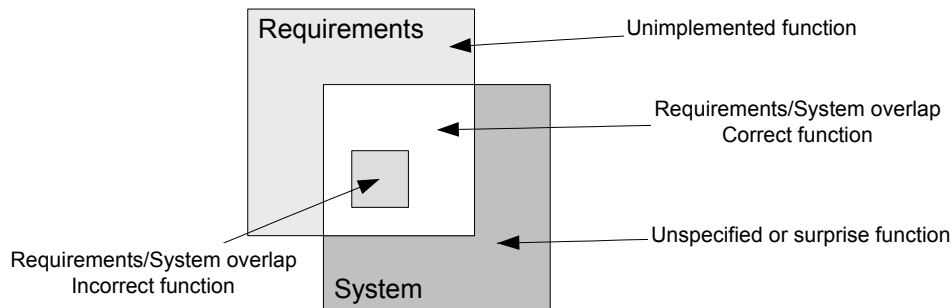


Figure 18 How software faults occur

2.8.5 Test strategies for functional testing

As shown in Figure 19, there are two opposing functional test strategies:

- **White-box or implementation-based testing**, where the tester has access to the source code. As a result, the tester has the **knowledge necessary to access every line of code** within the software under test, control the sequence in which code is executed, and can use this knowledge to design tests. White-box testing may be carried out during unit, integration or system testing.
- **Black-box, requirements-based, responsibility-based or functional testing**, where the tester has no knowledge of the code implementation. Instead, the tester relies on the **specified behaviour** or **responsibilities** to design tests. While black-box testing may be carried out during unit testing, it is more commonly carried out at integration and system testing.

Of the two test strategies, black-box testing is much more in keeping with the object-oriented paradigm and Design by Contract. The object-oriented paradigm uses the concept of encapsulation – while a class’ interfaces are defined (i.e. the behaviours), the inner workings of the class are hidden. Design by Contract uses assertions to express the responsibilities of both the Client and the Supplier.

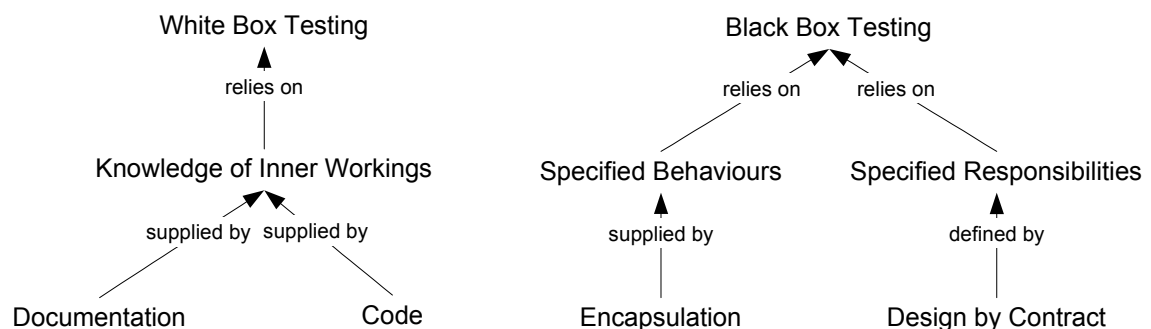


Figure 19 White Box Testing vs Black Box Testing

Regardless of the style of testing, covering all permutations and combinations in a system is limited (and so prevented) by:

- the high number of possible input and output combinations (white-box and black-box), and
- the high number of unique execution sequences (white-box),

As Dijkstra observed [39]:

... program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence.

The only way to a fault free system is to test every piece of functionality under every possible operational condition [29]. In reality, this is impossible!

2.8.6 Testing and the object-oriented paradigm

Object-oriented programming leads to more modular software, with the potential for ‘plug and play’ reuse. However, the hoped for improvements in efficiency have not been fully realised [38]. And, testing is still necessary.

Object-oriented programming generates its own **test issues**. Within the scope of this discussion:

- Object-oriented programming is very **dependent on the interactions** between the objects. Although a class is the smallest standalone code entity, a single class serves no useful purpose. Classes must be combined to make useful programs. As a result, interfaces are key, and must be tested to an acceptable level within each new implementation.
- While classes are designed for reuse, it is difficult for a:
 - class’ developer to **envisage all the situations** under which a class may be used, and
 - future users to **understand the exact thinking** behind the initial implementation.Testing for one application may not cover a class’ use in a different application. Testing must be repeated within the new situation.
- **Faults may be hidden**. A routine may appear to operate correctly. However, the object itself may be left in a degraded, corrupted state that only becomes apparent later, when another routine call is made. Routine sequences must be systematically identified and exercised to find hidden faults.
- Where there are **sequential calls** through a number of object instantiated from different classes, there is the increased potential for faults and hidden faults. These faults can only be identified through identifying and exercising these sequential calls.
- Where **code coverage** is an issue, encapsulation may hinder efficient working, by making it difficult to accurately access a specific line of code.
- Where **sequence of operation** is an issue, encapsulation may hinder efficient working, by making it difficult to accurately perform actions in a specific sequence.
- Classes are generally developed over a period of time, with earlier classes being tested as soon as they are developed. As a result, **testing is much more incremental**, and will have to take into account not only the addition of new classes, but also changes in existing classes, and the removal of classes that are viewed as unnecessary or obsolete.

2.8.7 Testing and Design by Contract

Design by Contract makes use of assertions to define pre-routine call states and post-routine system states [38]. While also helping to prevent faults being included in code in the first instance, and assist in debugging, the **assertions can be used to support testing** by being used to check:

- implementation-specific assumptions,
- implementation-independent assumptions, where the responsibilities of other items tested are not necessary documented, and
- relationships that must hold at routine entry and routine exit, so defining the range of test cases,

since they define:

- the acceptable bounds for a routine's input, and
- the acceptable bounds for a system once the routine has terminated.

Effectively, the assertions (if switched on at run-time) act as built in tests. However, they will not cover all testing needs, and cannot be relied on as the sole source of testing.

There are limits. Assertions may not have been defined, or may not have been fully defined. While it may be possible to identify and make explicit undefined and implicit assertions by studying behaviour models, there will always be doubt as to the completeness of the assertion set. Where an assertion set is not complete, the assertions may be counter productive with the benefits described in Section 2.1.6 not being fully realised (e.g. better documentation), or being falsely realised (e.g. if testing is based on incomplete assertions, the testing will be incomplete).

2.8.8 Conclusion

Testing is carried out to ensure that the requirements of a system or component have been met. Testing can be carried out at a number of levels within a system (e.g. unit testing, integration testing, and system testing). While functional testing checks functionality and error management, non-functional testing covers a wide range of properties (e.g. usability, maintainability, and portability). White box testing is carried out with full knowledge of the inner working of the item under test. Black box testing requires no knowledge of the implementation.

Within the object-oriented paradigm, interfaces are key, and must be tested to an acceptable level within each new implementation. The assertions within Design by Contract define the bounds of a routine's input and output. As a result, the contracts can be used as a basis for developing tests.

2.9 Regression Testing

Section 2.8 described testing in general, with particular focus on the testing that is carried out when a system is first developed.

Following initial testing of a system, further testing is generally carried out after each update to the system, to ensure that functionality that should not have changed has not changed during the update (i.e. the system has not regressed). This section covers this regression testing.

2.9.1 What is regression testing?

Regression testing has been described as [41][44]:

Selective retesting of a system or component to verify that modifications have not caused unintended effects and that the system or component still complies with its specific requirements.

Regression testing aims to revalidate the old functionality inherited from the predecessor system. The new version of the system should behave exactly as before, except for where a change in behaviour was intended.

Regression testing is generally done [38]:

- Following **bug fix**, to check that the bug has actually been fixed.
- Following **bug fix**, to check that no new bugs have been added.
- During **iterative developments** (i.e. build verification testing or smoke testing), after debugging of the new code, to ensure that the pre-existing functionality has not regressed.

- When a **reusable component** has been introduced to an existing system, to ensure that the reusable component has not broken pre-existing parts of the system.
- As the **first step** in integration testing and system testing, where updated components are included in the build, to ensure that the updated components have not broken the system.
- When **maintenance work** has been carried out.
- For **compatibility assessment and configuration testing**, when an application is ported to a new platform.
- For **benchmarking**.

Regression testing is particularly important in

- object-oriented development, where iterative development is the norm, and
- in any project where there are changes and error corrections, since this type of change tends to be more prone to error than the original development work [43].

Adequate retesting of a modified component only gives reason for confidence in the modified component. It does not give reasons for confidence at the integration or system level, since interactions may have changed. As a result, regression testing should be carried out at the unit test, integration test and system test level.

Regression testing does not reduce the need for new tests. Regression testing makes use of existing tests for functionality that should not have changed. New tests are still needed to cover existing functionality that has modified, and new functionality added since the last round of testing.

2.9.2 Why do regression testing?

Once a component has been updated, tested, and debugged, the original component within a system can be replaced by the updated component [38][44]. At this point, regression testing is needed to ensure that the modified parts of the system work correctly, and the unmodified parts of the system continue to work correctly, since small changes in one part of the system may have surprising knock-on effects elsewhere.

Possible faults introduced into an object-oriented system include:

- **Side effects.** The updated item may have a side effect on the system, unexpectedly allocating, de-allocating or changing the value of global variables, object variables, or database tuples. The system then fails because the updated component is not matched to the system's requirements, assumptions or contracts.
- **Violated pre-conditions or invariants.** The updated component may call a routine within the system. The routine call may violate an invariant or pre-condition within the system.
- **Violated post-conditions or invariants.** The system may call a routine within the updated component. If post-conditions within the updated component have changed or are incorrect, then the updated component may return a value to the system that violates a system invariant or post-condition. The system may then fail, or return an invalid value to an external system.
- **Data storage problems.** The updated component may produce an output that is in keeping with its specification. However, on saving to persistent storage, the value may trigger an existing fault in the system, or violate a contract with the system or an external system.
- **Compatibility issues.** The updated component may not be compatible with the system. Although the requirements and implementation have not changed, another change may lead to a failure (e.g. a hardware change).
- **Undesirable interactions.** An undesirable interaction may occur between the updated component and the system.

2.9.3 The regression test set

Regression testing reuses tests that have already been used for the initial testing [42]. As shown in Figure 20:

- A stable system will have a set of tests associated with it. When the tests are run, all the tests pass.
- If a like-for-like change is made to the system, where there is no change in the system's requirements, or component interfaces, there should be no change to the tests. The tests previously used to test the updated area of the system, can be reused within the **regression test set**.
- Following an update to the stable system, that changes the system's requirements, the original test set may contain tests that will not run or will fail when executed against the updated system.
- The new test set will contain:
 - original tests that are expected to continue to pass when run against the updated system, and form the **regression test set**,
 - tests that are no longer relevant, because of changes in the updated system's behaviour, and
 - new tests, generated to cover new behaviour within the updated system.

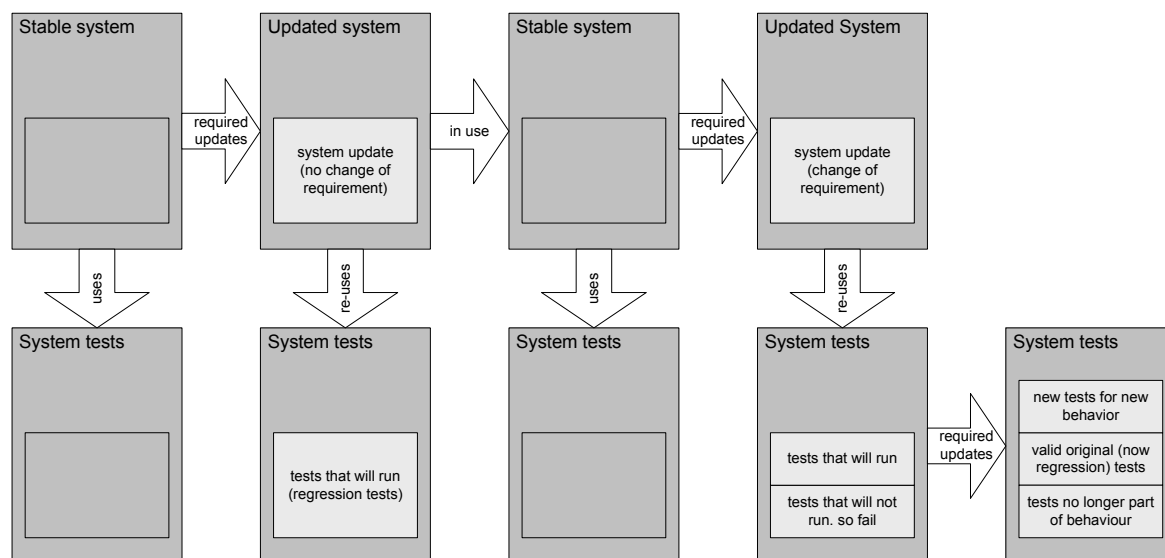


Figure 20 Tests and regression tests

2.9.4 Identification of the regression test set

Remarkably little is written on the processes that might be used to simplify and speed the selection of regression tests, as a means of reducing the regression test overhead. Instead, the focus is generally on the need to carry out regression testing, followed by a glib statement to 'select regression tests, and then run regression tests'!

There are two fundamental approaches to the selection of regression tests [51]:

- **Do not select** regression tests. Instead, re-run every test in the initial test suit. However, this approach becomes increasingly time and resource expensive as the size and complexity of the system grows since the reason for any test failures must be justified.
- **Apply selection heuristics** to the existing test set, so that only a targeted sub-set of the original test set is run. Typical regression test selection heuristics include focusing on:
 - risky functionality, where failure will result in significant loss (within the context of business or safety related loss).
 - frequently used functionality.

- directly testing the changed unit of functionality, external to the system within which the functionality sits.
- indirectly testing the changed unit of functionality, through 'end to end' testing, within the system, or
- random testing.

In both cases, there is a need to identify and remove obsolete test cases, since these tests do not form part of the regression test set. In both cases, it is difficult to justify that all the regression tests have been identified, and there are no false positive results.

The trade-off is:

- the cost of selecting and running the regression test set, against
- the ability to detect unintended side effects, following system update.

2.9.5 Conclusion

Regression testing is carried out to ensure that a system update has not had unwanted side effects. The regression test set is drawn from tests that have already been run, and have passed, on the previous version of the system. Identifying regression tests can be resource intensive and costly. As a result, heuristics are often used to identify suitable regression tests. However, such a regression test identification strategy may leave areas of the system untested, and can be difficult to justify within safety cases for safety critical systems.

2.10 Safety Cases and Goal Structured Notation

Safety cases present an argument that a system is acceptably safe. GSN is used to increase the clarity of the argument. This section describes the purpose of a safety case, and how GSN may be used to support a safety case argument.

2.10.1 Safety cases

The purpose of a safety case (together with supporting evidence) is to [52]:

... communicate a clear, comprehensive and defensible argument that a system is acceptably safety to operate in a particular context.

The use of safety cases has been widely adopted in industries where the safe operation of a system or system of systems is necessary to prevent loss due to injury or death.

Any safety case contains three essential elements:

- **requirements**, to express what the system must achieve to be acceptably safe,
- **arguments**, to explain how it can be concluded that the requirements have been addressed, and the system is acceptably safe, and
- **evidence**, to support the arguments being made.

However, for most real world systems, safety cases need to present large amounts of information, and can be very complex. As a result, safety cases can be difficult to assimilate. Arguments become lost in the detail, or are unfit for purpose.

2.10.2 Goal Structured Notation

GSN has been identified as a way of increasing the clarity of safety case presentation. This is achieved by graphically representing the individual elements within a safety argument, and making clear the

relationships that exist between these elements. Figure 21 demonstrates the GSN symbols that have been used within this document.

In essence:

- A **goal** is a proposition, about something that the argument wishes to demonstrate is true. Goals are the requirements within a safety case.
- Goals can be reduced in size, and argued, through the identification and development of **sub-goals**. Conversely, sub-goals can be brought together to support the argument stated in a goal. Reducing a goal to a number of sub-goals is a 'divide and conquer' strategy.
- **Context** is added to provide the clarification necessary for a goal to be clear and well defined.
- A **strategy** provides additional information about the way in which a goal will be addressed. Strategies are used to clarify how a goal will be achieved, particularly where there may be more than one means of achieving the goal. Strategies are the arguments within a safety case.
- **Assumptions** describe the situations that must hold for a goal or strategy to form part of the overall argument.
- **Justifications** explain why a strategy will adequately support a stated goal.
- Evidence in support of the goals is identified through **solutions**. Solutions are the tangible end points within an argument.

The elements of GSN are linked together as a **goal structure** using:

- **Solved by** connectors to **vertically link** goals, strategies and solutions (the main argument thread) using.
- **In context of** connectors to **horizontally link** contexts, assumptions and justifications (added to support and clarify the main argument thread).

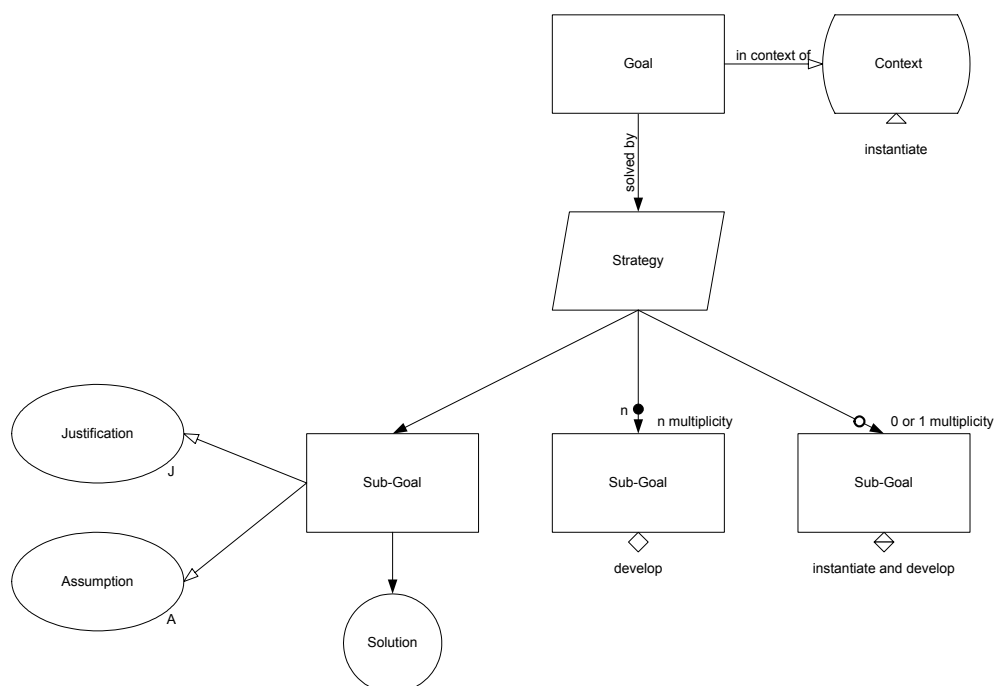


Figure 21 GSN pattern nomenclature

2.10.3 GSN patterns

Patterns can be used to describe common elements or repeating themes [52] [53]. For example, within the safety engineering domain, identical analysis may be carried out in a range of different situations (e.g. HAZOPs, ALARP, FMEA), and similar techniques may be used to increase the safety of a system (e.g. hazard removal, diverse implementations with voting). The use of these techniques is highly relevant within the context of safety case development, and will result in the repeated development of near

identical goal structures. The specific goal structures can be abstracted, to produce generic patterns that can then be tailored to a wide range of situations. The generic patterns can form a library of reusable GSN patterns, in a similar way to software being developed in the form of reusable components. GSN patterns can be used to embed knowledge and best practice, and prevent 'reinvention of the wheel'.

Where GSN patterns are developed, there may be unknowns that will only become apparent once the GSN pattern is applied in a specific situation (e.g. the number of hazards applicable to a system, the context for a specific goal, the types of evidence that might be used to support a goal). As shown in Figure 21, additional nomenclature is used to identify these unknowns:

- **Instantiate**, where a goal, context, strategy, assumption, justification of solution must be explicitly expressed.
- **Develop**, where a goal or strategy must be expanded upon, so putting in place the next layer of detail within the argument.
- **Develop and instantiate**, where an item must be explicitly expressed and further developed.
- **Multiplicity**, where 0, 1 or many instantiations of a goal may be necessary.

2.10.4 Modular GSN

GSN and GSN patterns have mainly been used to support safety arguments as stand-alone, single artefacts [56] [57] [58]. However, a large unit of GSN can be thought of and represented as a series of modularised, interconnected arguments. This is particularly the case where GSN is used to argue the safety of a modular system, where each of the modules within the system may already have a developed safety argument. It is the safety of the whole that needs to be argued, where the argument for the safety of individual parts of the system is coupled with the argument for the safety of the combined units.

Each of the modular arguments has the potential for reuse, either within the safety case argument for which it was developed, or within other safety case arguments (possibly outside the control of the module's original developers). Modular GSN is supported by extension to the core GSN notation. As shown in Figure 22, these extensions include:

- **Module Reference**, identifying a GSN module where a goal is defined and supported in another Module. The referenced module provides a reusable, modular solution to a particular goal.
- **Goals Identifier**, defining a goal, and identifying the module that provides a solution to that goal.

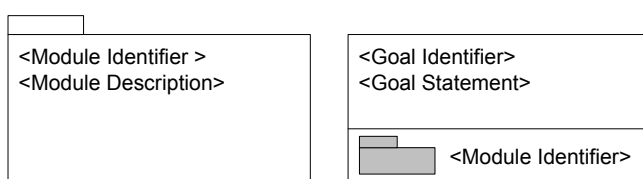


Figure 22 Modular GSN elements

2.10.5 Conclusion

A safety case must clearly, comprehensively, and defensibly argue that a system is safe to operate within a given context. A safety case contains three elements: requirements, arguments, and evidence. The amount of information to be presented can be considerable, making safety cases difficult to understand and assimilate. GSN has been applied to safety cases, as a means of hierarchically decomposing and diagrammatically representing the arguments that are being made, along with the contexts, strategies, assumptions, justifications and evidences that are being used to support the argument.

GSN patterns are developed as a way of recording common elements and repeating themes within safety case arguments. GSN modules are developed as a way of formalising and referencing discreet units of reusable GSN based argument.

2.11 Summary of literature survey

Modular systems, based on a mixture of off the shelf and bespoke components, are increasing being developed within both the non-safety and safety domains. Within the safety domain, a modular system must be accepted (and possibly certified). Generally this is done through the development of a safety case.

Modular systems tend to be incremental in nature. Development may be carried out in an incremental way, with core functionality being established, and then extended to include additional functionality. Additionally, as updated versions of existing modules become available, there are often pressures to replace the original modules with the newer versions.

As a modular system is updated, regression testing should be carried out to demonstrate that pre-existing functionality that should not have been broken by the update has not been broken by the update. In the safety domain, regression testing will also form part of the updated safety case, necessary before any updated system can 'go live'.

The identification of regression tests can be a resource intensive activity, which is both costly and time consuming. While there are strategies for identifying regression tests, these are often ill defined. As a result of this, heuristic techniques are often used to identify regression tests. However, for a safety case to be developed, it is necessary to be able to justify the selection of regression tests, and demonstrate that the unchanged pre-existing functionality has been fully tested.

This dissertation investigates the use of Design by Contract and safety contracts as a means to identify the regression test set that should be used following update of a modular safety critical or safety related system.

3 Problem statement and proposed approach

3.1 Problem statement

As discussed in Section 2.9, **regression testing** is used to check that when a change is made to a pre-existing unit of functionality the change does not result in any unexpected side effects (i.e. break functionality that should remain untouched). A regression test set is based on tests that have already been run on a previous version of the system under test, and have passed. By passing, the test has demonstrated that the targeted functionality within the item/items under test was acceptable, with respect to the facets that were being tested, at the time of testing. If, following a change, a test was again expected to pass (i.e. the functionality of interest should not have changed), and the test continues to pass, then the item under test has not regressed (i.e. there were no unexpected side effects as a result of the update). However, if a test fails, when it was expected to pass, then the item under test has regressed, and further investigation is required to understand and correct the situation.

Typically, regression testing is carried out:

- following bug fixes, at the unit and module level, and
- when units of functionality are substituted with alternative units of functionality, at the sub-system and system level (e.g. within modular systems).

As discussed in Section 2.9, **regression testing can be a significant overhead on a project**. In particular, the load that regression testing places on a project may be significantly out of proportion with the progress that the change causing the regression testing makes to a project (e.g. a single, simple change at the unit level may result in regression testing having to be carried out at the unit, module, sub-system and system level). For regression testing to be carried out, the regression test set must be identified, and then executed. To simplify the selection of regression tests, a heuristic is often used. For example, regression tests may be selected to focus on risk (e.g. regression test the functionality associated with the highest loss risk) or functionality (e.g. regression test the most frequently used functionality).

As discussed in Section 2.3, **modular systems are increasingly being used** for both non-safety critical applications and (more recently) for safety critical applications. One of the key drivers for modular systems is the ability to be able to use commercially available modular components, with the associated reduction in development risks. However, the components to be integrated will generally come from a number of different suppliers. As a result, there is no guarantee that the components will work together correctly. And, the components are generally 'black box', with the inner workings of the components not disclosed to system integrators and testers.

Initial testing of a safety critical modular system will provide proof of correct integration and operation, and provide supporting evidence for the safety case.

As newer versions of supplied modules become available, one or more of the original modules may be replaced with a newer version of the module – the updates may be necessary or desirable for a number of reasons, including fault correction, extension of functionality, and warranty considerations. In safety critical systems, **regression testing will be required to demonstrate that the incremented system continues to work correctly**, and to provide evidence for an update to the safety case. To carry out regression testing, existing tests that remain valid following an update must be identified. Any other tests, which will not form part of the regression test set, need to be either updated to fit within the context of the updated system, or discarded because they are no longer relevant. Additionally, new tests may be required to cover new functionality.

Modular systems may also be developed incrementally over a period of time, to provide enhanced or extended functionality. Again, regression testing will be required to demonstrate that the pre-existing part of the system continues to work correctly, and to provide evidence for an updated safety case.

Design by Contract was developed as a means of managing complexity, by viewing a system as a set of components, where the component interactions are defined within mutually obliging contracts. As described in Section 2.1, Design by Contract provides support for testing and component reuse. As demonstrated in Section 2.2, while Design by Contract was initially developed for software systems, the

ideas presented within Design by Contract have been extended for use in higher level system modelling. Section 2.5 extends the contracts from within Design by Contract to cover the wider dependencies within systems, through the development of safety contracts.

The question is:

Can safety contracts and the Design by Contract rules relating to modular interchange be used to guide the identification of:

- **pre-existing tests that form the regression test set,**
- **pre-existing tests that need to be updated,**
- **pre-existing tests that need to be discarded, and**
- **areas of new functionality within the updated system, for which new tests will be required**

following update to a modular system?

3.2 The approach

Design by Contract can be used to specify contracts between a Client and a Supplier for each unit of functionality that the Supplier makes available to the Client. However, in Design by Contract, the contracts are focused on the relationship between the Client and the Supplier, and do not extend to cover the wider system dependencies.

As described in Section 2.5, **rely – guarantee** reasoning can be used to extend Design by Contract into a system context, by defining contracts between the Supplier and other units of functionality within the system, at the time that a Client calls that functionality, and after the functionality has terminated.

As described in Section 2.7, the concept of **safety contracts** combines Design by Contract with system based rely/guarantee reasoning. The relationships between modules acting in a Client capacity, and the modules acting in a Supplier capacity are identified for a system, along with the pre-conditions and rely conditions that must hold prior to a routine call from a Client to a Supplier, and the post-conditions and guarantee conditions that will hold after the routine call has terminated.

The **rules covering the interchange** of units of functionality within systems developed using Design by Contract are discussed in Section 2.6. These rules extend to include the additional rely conditions and guarantee conditions contained within safety contracts.

The aim of this work is to evaluate the use of safety contracts as a means of identifying tests that remain unchanged following updates being made to a modular system. These tests will form the regression test set.

4 The working example

4.1 Active steering – an example of a safety critical system

Generally, a system comprises a number of units of functionality (referred to as sub-systems, components or modules), held together by interfaces. However, on further analysis, the sub-systems can be broken down further, into finer granules of functionality, held together by more interfaces. This decomposition is part of the ‘divide and conquer’ technique used to identify a system’s fundamental components and interfaces. Decomposition continues until an area of functionality is sufficiently atomic and well defined, so that the functionality and interfaces can be adequately specified, such that they can be built.

In a modular system, the higher-level units of functionality are often standalone units, which can be bought off the shelf, and will be applicable to a range of systems and situations. For example, in a typical automotive active steering system, the high level system might comprise:

- a traditional, mechanical, **rack and pinion steering system**,
- servo assistance, via
 - a **hydraulics reservoir**,
 - a **valve**, and
 - a **steering pump**,
- a system to increase or decrease the command from the steering wheel to the road wheels, via
 - an **electronically controlled gear box**, and
 - an **actuator**,
- a range of **sensors** to monitor the behaviour of various key components within the car, typically including:
 - an actuator sensor (measures angle applied via the electronically controlled gear box),
 - road wheel sensor (measures road wheel angle from true, and revolutions per minute), and
 - a steering wheel sensor (measures steering wheel angle from true)
- an **electronic control unit** (ECU),
- **wiring** to carry data from the sensors to the ECU, and from the ECU to the actuator for the electronically controlled gear box,
- **power supplies** for various electrical elements within the system, and
- a **CAN bus** to carry data between the active steering system and other systems within the vehicle.

The system is shown diagrammatically in Figure 23.

At the sub-system level, the functionality becomes more specialised and specific. For example:

- a **rack and pinion steering system** reduces to the steering rack, the pinion, the drive shaft, the steering rods, etc., and
- the **ECU** reduces to processors, electronic components, software to support system monitoring (signal input from sensors) and software to support system command (signal output to actuators)

At the sub-sub-system level, further decomposition can then be carried out. For example:

- the **steering rack** will reduce to a number of components, including the shaft itself and the associated steering rods, and
- the **system command software** will include a number of software modules to compare inputs, calculate outputs, carry out comparisons when more than one independent process stream is used (x out of y voting).

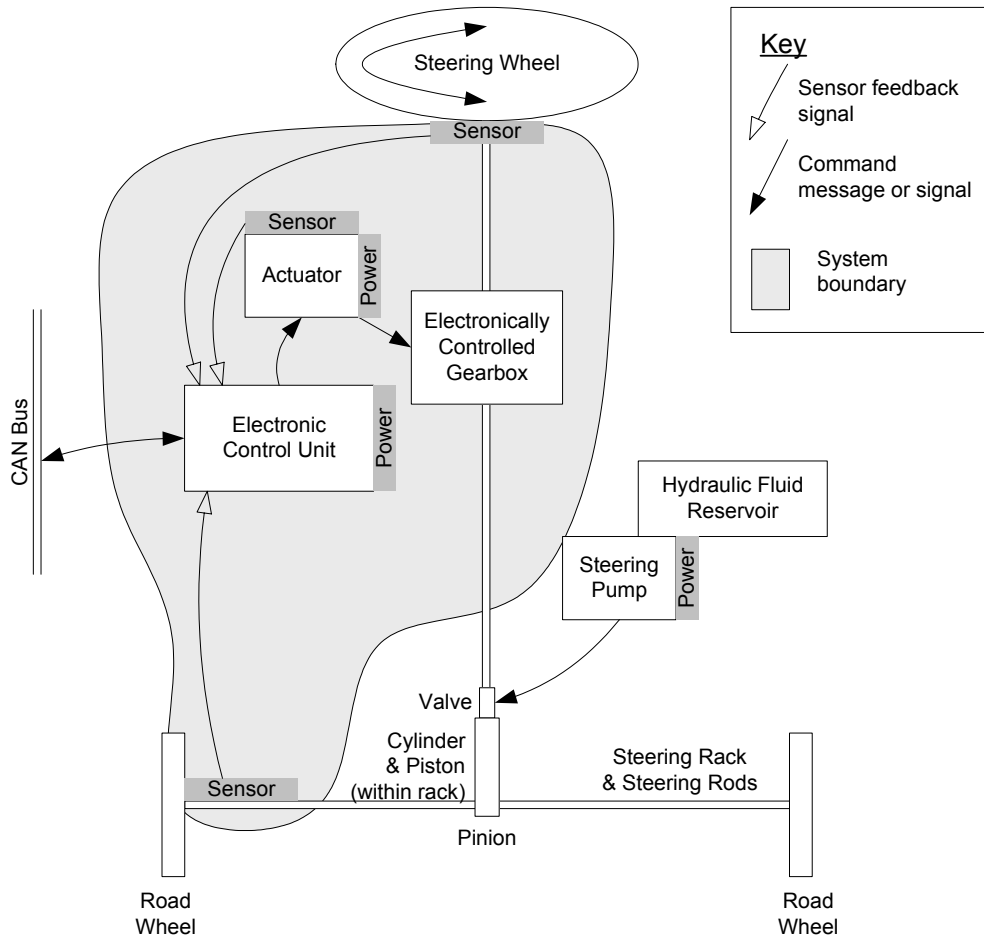


Figure 23 High-level system view of active steering system

At each level of the decomposition, the principle is the same:

- a unit of functionality interacts with other units of functionality, via its interfaces, and
- pre-conditions, post-conditions, rely conditions and guarantee conditions can be specified for the routine calls.

While the scenarios used below are at the code module level, the basic principle is the same at any level within the system of systems hierarchy. A system of systems exists at a number of levels, and safety contracts can be specified at any level in the system hierarchy. The hierarchical system-of-systems scenario is shown diagrammatically in Figure 24.

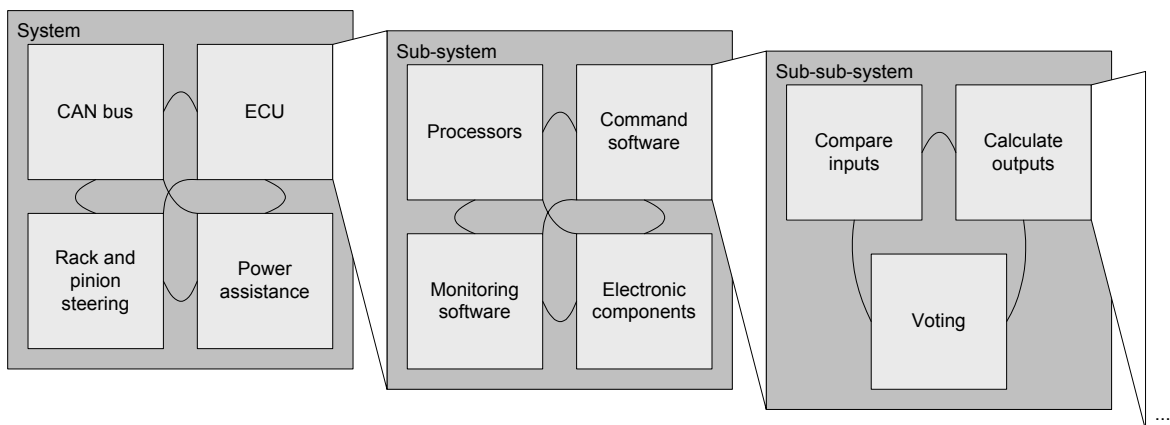


Figure 24 Systems, sub-systems and functional units within an active steering system

This work will focus on the sub-system level, on the software within the ECU used to create the signal to command the actuator, thereby increasing or decreasing the road wheel effect of turning the steering wheel.

4.2 Working definitions

A **safety contract** comprises:

pre-condition AND rely condition AND post-condition AND guarantee condition

Safety contract clause:

Any single predicate statement within a safety contract

A **pre-contract** comprises:

pre-condition AND rely condition

A **post-contract** comprises:

post-condition AND guarantee condition

A **safety contract condition** comprises:

pre-condition OR rely condition OR post-condition OR guarantee condition

4.3 Assumptions

Systems are inherently complex. As a result, it can be difficult (if not impossible) to think about a system in its entirety, with all the complexities in place. This is true even for the relatively simple scenario described here. As a result, it has been necessary to simplify the problem, so that at least a part of the problem can be productively worked with, and some of the possible solutions can be found. For the analysis carried out in Section 5 and Section 6, the following assumptions have been made, as a means of reducing the problem to an initially surmountable size, so that an initial evaluation can be carried out.

It is assumed that:

- **Assumption 1:** The safety contracts identified between the Clients and the Suppliers are based on integer clauses. This is further expanded on in Section 5.

This simplifies the safety contracts developed in Section 5, and used in the main body of this work, by reducing the complexity of the changes that are possible within a safety contract.

- **Assumption 2:** For each function or procedure, the range of acceptable values for all the input variables passed from the Client to the Supplier is fully specified in the pre-contract.
- **Assumption 3:** For each function, the range of possible values for all the variables returned from the Supplier to the Client is fully specified in the post-contract.
- **Assumption 4:** For each procedure, the range of possible values for the Supplier updated variables is fully specified in the post-contract.

Assumptions 2, Assumption 3 and Assumption 4 are basic good practice. These assumptions ensure that the component interface is fully specified, as is required to take full benefit from Design by Contract. These assumptions allow the full range of test conditions to be identified, where testing is being based on the safety contracts. These assumptions also ensure that that the comparisons being made between an original safety contract and an updated safety contract do not have to consider the consequences that might arise as the result of incomplete safety contracts. These assumptions form the basis of future work identified in Section 11.7.

- **Assumption 5:** Prior to a change being made to a modular system, the original system was tested , and the original set of tests that passed is identified.

This provides a system that has been baselined, against which comparisons can be made following regression testing. This also establishes tests that are functionally correct, with known results. This means that missing tests, or irrelevant tests do not have to be considered, and provides an identified set of tests from which the regression tests must be drawn.

- **Assumption 6:** Initial testing has been carried out within the context of the safety contract, with pre-contracts being used to identify valid routine calling conditions and the post-contracts being used to identify valid post-routine conditions.

This work is based on system testing via safety contracts. As a result, the initial testing has to be based on safety contracts.

- **Assumption 7:** Initial testing has been carried out to exercise the full range of the safety contract (i.e. extreme limits, as well as the (probably more usual) median cases).

Again, this is basic good practice. This assumption ensures that that the test and regression test result comparisons do not have to take gaps in the testing into account.

- **Assumption 8:** Any new, updated module has been tested to an 'acceptable' standard, at the level below the current level of testing. (In this example, the level below the current level of testing will be the sub-sub-system testing.) The meaning of 'acceptable' has been defined and recorded.

The next step in the testing chain is integration testing, where any new module is included within the system under test. (In this example, the current level of testing will be the sub-system integration testing.)

By testing the updated module prior to integration testing at the next level, the systems of systems within the updated module and the resulting interfaces within the updated module have been tested. Any faults that are identified when the updated module is integrated at the next level up are likely to occur as the result of the new interactions and interfaces that are generated with that next level of integration. It is these higher level interactions and interfaces that will be directly tested via their safety contracts. This additional testing simplifies the test target.

- **Assumption 9:** The system is sequential, and does not use parallel processes, with multiple updates occurring within the system.

Parallel processing implies that the satisfaction of the rely conditions and the guarantee conditions may not hold for the whole of the time that a routine call is being executed, as a result of the side effects of other routines being executed in other parts of the system. This assumption removes the complexities that parallel processing within the system may cause. This assumption forms the basis of future work identified in Section 11.7.

5 Safety contracts and regression testing

5.1 Integer based safety contracts in a typical safety critical system

Safety contracts can be expressed in terms of primitive types or composite types. Ultimately, all composite types can be reduced to, and defined by, primitive types. Typical primitive types include:

- numbers (integer, real, and the associated variants),
- strings, and
- booleans.

This dissertation will initially focus on integer based safety contracts within the Supplier.

In particular, the software within the ECU receives a number of input values from sensors within the vehicle, and provides an output signal that is used to control the active steering actuator, and so provide the active steering functionality.

Provided the input feedback signals received from the sensors and passed to the control software are within specific ranges, the output value used to control the actuator will also be within a specific range. These value ranges are critical to the determination of the control command for the active steering gearbox, and should be defined within the control software pre-conditions and the post-conditions.

In addition, to operate correctly and safely, this software also relies on the battery voltage being between 10 Volts and 14 Volts (i.e. from a (nominally) 12 volt battery), with monitoring being carried out within the ECU.

Provided the pre-condition (states within the modules) and rely condition (states within the wider system) hold, the command software guarantees that the actuator will deliver a correct and controlled (i.e. smooth, without jumping) turning on the steering column. If the active steering is not delivered in a controlled manner, a hazardous situation is established, that may lead to loss of life or injury to the driver.

A number of integer-based clauses have been identified in the Supplier's safety contract, as follows:

```
Supplier safety contract for the function COMMAND
  Pre contract
    Pre-condition
      -180 <= steering_wheel_angle <= 180 [degrees]
      vehicle_speed >= 1 [km/hour]
      steering_flag = 1 [state definition]
      ...
    Rely conditions
      10 <= battery_voltage <= 14 [Volts]
      ...
  Post contract
    Post-condition
      0 <= command <= 20 [power to be supplied]
      (steering_state = 0) or (steering_state = 1) or (steering_state = 2)
      [state definition]
      ...
    Guarantee conditions
      0 <= actuator_rate_of_change <= 10 [degrees per second]
      ...
```

The active steering functionality is called from a controlling (Client) function, hard coded within the ECU.

5.2 Changes to integer based safety contracts

When a module is updated, integer based pre-contracts and post-contracts may change in a number of ways. The pre-contracts and post-contracts may become:

- **Stronger**, or
- **Weaker**.

Alternatively, the changes may fall outside of the stronger/weaker definition, and be classified as **complex**. This section examines these concepts in more detail.

5.2.1 Stronger pre-contracts and post-contracts

Stronger pre-contracts or post-occur when:

- One or more of the existing clauses in a pre-contract or a post-contract becomes **more restricted**, (i.e. the new safety contract clause is contained within the old safety contract clause), or
- One or more additional, **new clauses are added** to a pre-contract or a post-contract, or
- Both of these changes can occur within a pre-contract or a post-contract.

For example, an existing safety contract condition:

```
Pre-condition
    -180 <= steering_wheel_angle <= 180
```

is strengthened when it becomes more numerically restrictive:

```
Pre-condition
    -100 <= steering_wheel_angle <= 100
```

or when an additional clause is added:

```
Pre-condition
    -180 <= steering_wheel_angle <= 180
    vehicle_speed >= 1
```

5.2.2 Weaker pre-contracts and post-contracts

Weaker pre-contracts or post-occur when:

- One or more of an existing pre-contract or post-contract clauses becomes **less restrictive** (i.e. the old safety contract condition is contained within the new safety contract condition), or
- One or more of the **existing clause are removed** from a pre-contract or a post-contract, or
- Both of these changes occur within a pre-contract or a post-contract.

For example, an existing safety contract condition:

```
Pre-condition
    -180 <= steering_wheel_angle <= 180
    vehicle_speed >= 1
```

is weaker when it becomes less numerically restrictive:

```
Pre-condition
    -360 <= steering_wheel_angle <= 360
    vehicle_speed >= 1
```

or when an existing clause is removed:

```
Pre-condition
    vehicle_speed >= 1
```

5.2.3 Complex changes to pre-contracts and post-contracts

Complex changes to pre-contracts and post-contracts occur when one or more of the existing clauses in a pre-contract or post-contract changes, such that the change cannot be described as more or less restrictive. Examples include:

- The addition of new clauses and removal of existing clauses within the same pre-contractor or post-contract, or
- An updated pre-contract or post-contract clause overlaps the original clause, or is disjoint from the original safety contract clause).

For example:

```
Pre-condition
    -180 <= steering_wheel_angle <= 180
```

is a complex change when it becomes

```
Pre-condition
    -20 <= road_wheel_angle <= 360
```

since the old pre-condition is not contained in the new pre-condition, and the old pre-condition does not contain the new pre-condition. Instead, there is a partial overlap between the old pre-condition and the new pre-condition.

5.3 Conclusion

This section has:

- Described an integer based safety contract as one where the clauses within the safety contract contain only integer relationships within the assertions.
- Described how pre-contracts and post-contracts may be changed, to make a:
 - stronger contract,
 - weaker contract, or
 - contract change that cannot be classified as stronger or weaker, so has been classified as a complex change.

The work presented in this section is used in Section 6 to consider how safety contracts can be used to identify regression tests in a two module Client – Supplier relationship, where the safety contracts are based on integer clauses.

6 Supplier regression testing via integer safety contracts

Scenario 1 will focus on a two module Client – Supplier relationship, where the Supplier is updated. The Client is the system control software in the ECU. The Supplier is the actuator control software. The scenario is shown diagrammatically in Figure 25. In particular, the focus is on a function and its associated safety contract specified in the Supplier’s interface as being available from the Supplier, and the tests used to test that function (e.g. the COMMAND function, the COMMAND function safety contract and the tests used to test this function in both the original and updated system).

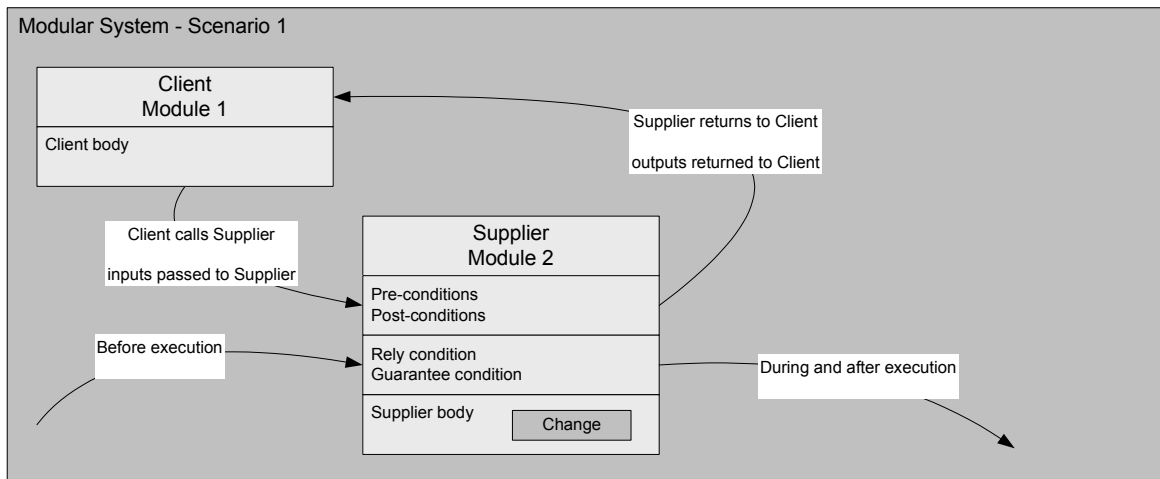


Figure 25 Scenario 1 – A two module Client/Supplier system, with change in the Supplier

In Scenario 1, the question is:

Under what circumstances of change to a function provided by the Supplier will the Client and the system still:

- satisfy the updated Supplier function pre-contract, and
- be satisfied by the updated Supplier function post contract?

If these circumstances are satisfied, the original test set associated with the function becomes the regression test set. When these circumstances are not satisfied, the original tests will not form part of the regression test set. Instead, the existing tests will need to be updated (to cover pre-existing functionality where the pre-contracts and post-contracts have changed), discarded (where the functionality is no longer applicable), or new tests will be required to cover new functionality within the system.

Once the regression tests have been identified, they can be executed, to establish if the system has or has not regressed.

6.1 Scenario 1a – no change to Supplier’s safety contract

In this situation, the inner working of the Supplier module may have changed, but the safety contract associated with the function is unchanged. Typically, this could happen at the code level when a patch is applied, or at the ‘system of systems’ level when a new version of one of the systems within the system is released (e.g. a new version of a database). However, the interface conditions, in the form of the safety contract, remain unchanged.

6.1.1 Analysis

As discussed In Section 2.1:

- Encapsulation and Design by Contract means that we are only concerned with the interfaces, and in particular, with the contracts defined within the interfaces. While encapsulation and Design by Contract were originally envisaged within the context of code modules within object-oriented programmes, the same principles can be applied at any level within a system.
- Support for code reuse, or substitution of one item with another, is provided via the satisfaction of the interface definitions (i.e. the pre-contracts and post-contracts defined within a safety contract).

As discussed in Section 2.6:

- If the pre-conditions and the post-conditions for a function hold, then both contracts are maintained and the Supplier will execute that function without error (assuming the updated Supplier's internal functionality executes correctly – this should be the case if the updated Supplier has been adequately tested).

When a function's safety contract is unchanged following an update to a Supplier, this is (in effect) a like for like change to the Supplier functionality. The Supplier does not change the requirements placed on the Client or the system, with respect to the values input by the Client or the state of the system. And, the values returned by the Supplier to the Client and the system will continue to be in the same range as previously defined.

6.1.2 Regression test set

Since the safety contract associated with a function specified within the Supplier's interfaces is unchanged, when a test based in the original functionality/safety contract has passed, it should continue to pass following update to the inner workings of the Supplier.

6.2 Scenario 1b – no change to Supplier's pre-contract

In this situation the function within the original Supplier, and the updated Supplier, have the same pre-contract.

6.2.1 Analysis

This situation is similar to Scenario 1a. However, only the function's pre-contract remains unchanged – the post contract has changed in some way.

If a pre-contract remains unchanged, then the function's valid/invalid inputs from the Client remain unchanged, and the function's valid/invalid system states remain unchanged.

6.2.2 Regression test set

Any test based on the function's original pre-contract must continue to be valid with respect to the pre-contract following update, if the pre-contract does not change at update. However, the changes to the function's post-contract will need to be evaluated (see Section 6.7 and Section 6.9) before the regression test set can be fully established.

6.3 Scenario 1c – weaker Supplier pre-contract

In this situation the function within the updated Supplier has a weaker pre-contract, when compared with the function within the original Supplier pre-contract.

6.3.1 Analysis

If a function's pre-contract clause is **removed**, that part of the safety contract becomes TRUE under all circumstances (removing the clause, and so removing the restrictions on the integers defined in that

clause, indicates that the clause's restrictions are no longer of interest). Any part of a test based on a removed pre-contract clause will remain valid following update of the Supplier.

If a function's pre-contract clause is **relaxed**, the rules described in Section 2.6 can be applied. Since the new pre-contract clause is broader, but continues to contain the conditions specified in the original pre-contract clause, any test condition based on the old pre-contract clause will remain valid with the new pre-contract clause.

6.3.2 Regression test set

Tests based on the function's original stronger pre-contract will continue to be valid with an updated weaker pre-contract. As a result, tests based on the function's original stronger pre-contract may form part of the regression test set. However, any changes to the function's post-contract must be evaluated before the regression test set can be fully established.

6.4 Scenario 1d – stronger Supplier pre-contract

In this situation the function within the updated Supplier has a stronger pre-contract, when compared with the original function pre-contract.

6.4.1 Analysis

If a pre-contract clause is **added** to a function, the Supplier's function makes further (unexpected within the context of the initial test set) demands on any input from the Client. Any test written against the original pre-contract cannot be guaranteed to be valid with the new pre-contract in place.

If a function's pre-contract clause is **strengthened**, the rules described in Section 2.6 can be applied. Since the function's new pre-contract clause is narrower, it will not continue to contain all the values specified in the original pre-contract clause. Any test written against the function's original pre-contract cannot be guaranteed to be sufficient with the function's new pre-contract in place, and will need updating to remain valid.

6.4.2 Regression test set

Since a function's stronger pre-contract will not continue to be valid with tests written for a weaker pre-contract, tests based on the weaker pre-contract will not form part of the regression test set.

6.5 Scenario 1e – complex change to Supplier's pre-contract

In this situation various changes have been made to the function's pre-contract in the updated Supplier. The sum total of the change is such that the function's updated pre-contract cannot be classified as either stronger or weaker than the function's original pre-contract.

6.5.1 Analysis

The changes to the function's pre-contract are too complex to apply the rules identified in Section 2.6 and applied to other scenarios within this Section.

6.5.2 Regression test set

None of the original tests can be assumed to remain valid after this type of change. This method cannot be used to identify regression tests. However, the functionality tested may remain valid. As a result, the

original test may form a starting point for update to cover the updated pre-contract. This topic forms the basis for future work, as identified in Section 11.7.

6.6 Scenario 1f – no change to Supplier’s post-contract

In this situation the function within the original Supplier, and the updated Supplier, have the same post-contract.

6.6.1 Analysis

This situation is similar to Scenario 1a. However, it is the function’s post-contract that remains unchanged – the function’s pre-contract has changed in some way. The values returned to the Client will continue to have the same range of values. Since the Client was able to handle the values returned by the original function post-contract, and the post-contract remains unchanged, then the Client will continue to be able to handle the values returned by the updated Supplier functionality.

6.6.2 Regression test set

Any test based on the original function’s post-contract must continue to be valid with the function’s post-contract following update, if the function’s post-condition does not change at update. However, the changes to the function’s pre-contract will need to be evaluated (see Section 6.3 and Section 6.5) before the regression test set can be fully established.

6.7 Scenario 1g – weaker Supplier post-contract

In this situation the function within the updated Supplier has a weaker post-contract, when compared with the function within the original Supplier post-contract.

6.7.1 Analysis

If a function’s post-contract clause is **removed**, that part of the safety contract becomes TRUE under all circumstances (removing the clause, and so removing the restrictions on the integers defined in that clause, indicates that the clause’s restrictions are no longer of interest). As a result, restrictions contained in the clause, relating to the range of values that the Supplier returns to the Client, are removed.

If a function’s post-contract clause is **relaxed**, the rules described in Section 2.6 can be applied. Since the new post-contract clause is broader, the original post-contract conditions continue to be contained in the new post-contract conditions. However, the Supplier may return additional variable values to the Client.

In both cases the Client has to be able to accept a broader range of values. This may not be the case.

6.7.2 Regression test set

Since the function’s weaker post-contract may return values that are not acceptable to the Client, tests based on the original stronger post-contract may not be valid with an updated weaker post-contract. Tests written for a stronger post-contract will not form part of the regression test set.

6.8 Scenario 1h – stronger Supplier post-contract

In this situation the function within the updated Supplier has a stronger post-contract, when compared with the original function post-contract.

6.8.1 Analysis

If a post-contract clause is **added** to a function, the variables defined within that part of the contract go from having an undefined range of values, to having a fixed range of values. In effect, the newly specified range will continue to fall within the original, unspecified range. If any value of the undefined variable was originally acceptable to the Client, then the value of the defined variable will also be acceptable to the Client. Any test based on the function's original post-contract will remain valid following this type of update to the Supplier.

If a function's pre-contract clause is **strengthened**, the rules described in Section 2.6 can be applied. Since the function's new post-contract clause is narrower, it will continue to be contained within the conditions specified in the original post-contract clause. If any value of the more broadly defined variable was originally acceptable to the Client, then the value of the more narrowly defined variable will also be acceptable to the Client. Any test condition based on the function's old post-contract clause will remain valid with the function's new post-contract clause.

6.8.2 Regression test set

Tests based on the function's original weaker post-contract will continue to be valid with an updated stronger post-contract. As a result, tests based on the function's original weaker post-contract may form part of the regression test set. However, any changes to the function's pre-contract must be evaluated before the regression test set can be fully established.

6.9 Scenario 1i – complex change to Supplier's post-contract

In this situation various alterations have been made to the function's post-contract in the updated Supplier. The sum total of the change is such that the function's updated post-contract cannot be classified as either stronger or weaker than the function's original post-contract.

6.9.1 Analysis

The changes to the function's post-contract are too complex to apply the rules identified in Section 2.6 and applied to other scenarios within this Section.

6.9.2 Regression test set

None of the original tests can be assumed to remain valid after this type of change. This method cannot be used to identify regression tests. However, the functionality tested may remain valid. As a result, the original test may form a starting point for update to cover the updated post-contract. This topic forms the basis for future work, as identified in Section 11.7.

6.10 Combining changes to pre-contracts and post-contracts

Section 6.2 through Section 6.9 looks at the localised effects of changing the pre-contract or post-contract on a Supplier's function, where the Client remains unchanged. However, if a safety contract based test that has been run and has passed prior to an update to the Supplier is to continue to be valid after an update to the Supplier (and so form part of the regression test set), then both the function's pre-contract and the function's post-contract must conform to the rules introduced in Section 2.6, and expanded on in Section 6.2 through Section 6.9. The changes (or lack of changes) to a function's pre-contracts and post-contracts have to be considered together.

Figure 26 combines the possible changes to a function's pre-contracts and post-contracts, where a Supplier provides the functionality that has been updated, but the Client remains unchanged. Cells marked '**Yes**' identify the combinations where, on the grounds of changes to a Supplier function's pre-

contract and Supplier function's post-contract, a test used to test the initial version of a Supplier function can be used as part of the regression test set following update to a Supplier function. For example, where the Supplier function has a weaker pre-contract, and there is no change to the function's post-contract, then the original tests for this function will form part of the regression test set. Conversely, where the Supplier function has a stronger pre-contract, and there is no change to the function's post-contract, then the original test for this function will not form part of the regression test set.

Nature of change to Supplier function	No change to pre-contract	Weaker pre-contract	Stronger pre-contract	Complex change to pre-contract
No change to post-contract	Yes	Yes	No	No
Weaker post-contract	No	No	No	No
Stronger post-contract	Yes	Yes	No	No
Complex change to post-contract	No	No	No	No

Figure 26 Validity of tests as regression tests based on pre-contract and post-contract changes

6.11 Conclusion

This section has developed rules for the identification of regression tests within modular systems, where:

- the functionality to be tested is called by a Client, and provided by a Supplier, in a two module Client-Supplier pairing,
- following initial testing to establish a test baseline, the Supplier has been updated, and
- the initial test set is based in safety contracts.

In keeping with the weaker pre-condition/stronger post-condition rule for inheritance within the context of object-oriented programming and Design by Contract, and within the limitations of this work, regression tests can be identified from the original test set where:

- the Client calling the Supplier is unchanged,

and

- the Supplier function's pre-contract is unchanged, and the Supplier function's post contract is unchanged or stronger, or
- the Supplier function's pre-contract is weaker, and the Supplier function's post contract is unchanged or stronger.

Section 7 describes a process (with examples) for applying these rules.

7 Process to identify regression tests via safety contracts

Section 6 provides an analysis of the way in which contract rules embedded within Design by Contract can be used to identify regression tests in a two module system, where:

- the Client is unchanged,
- the Supplier has changed as a result of some form of update, and
- the original test set is based on the Supplier’s safety contract.

This section looks at the process that should be followed to take advantage of safety contract based testing for the identification of regression tests. This is a cyclical process. Once a set of regression tests have been run and passed, the regression tests form part of the baselined, updated test set. When another cycle of change occurs, the process will be repeated on the updated test set (including the regression tests subsumed into the updated test set). The original regression test set may not form part of the future regression test set – each test must again be analysed to identify the new regression test set.

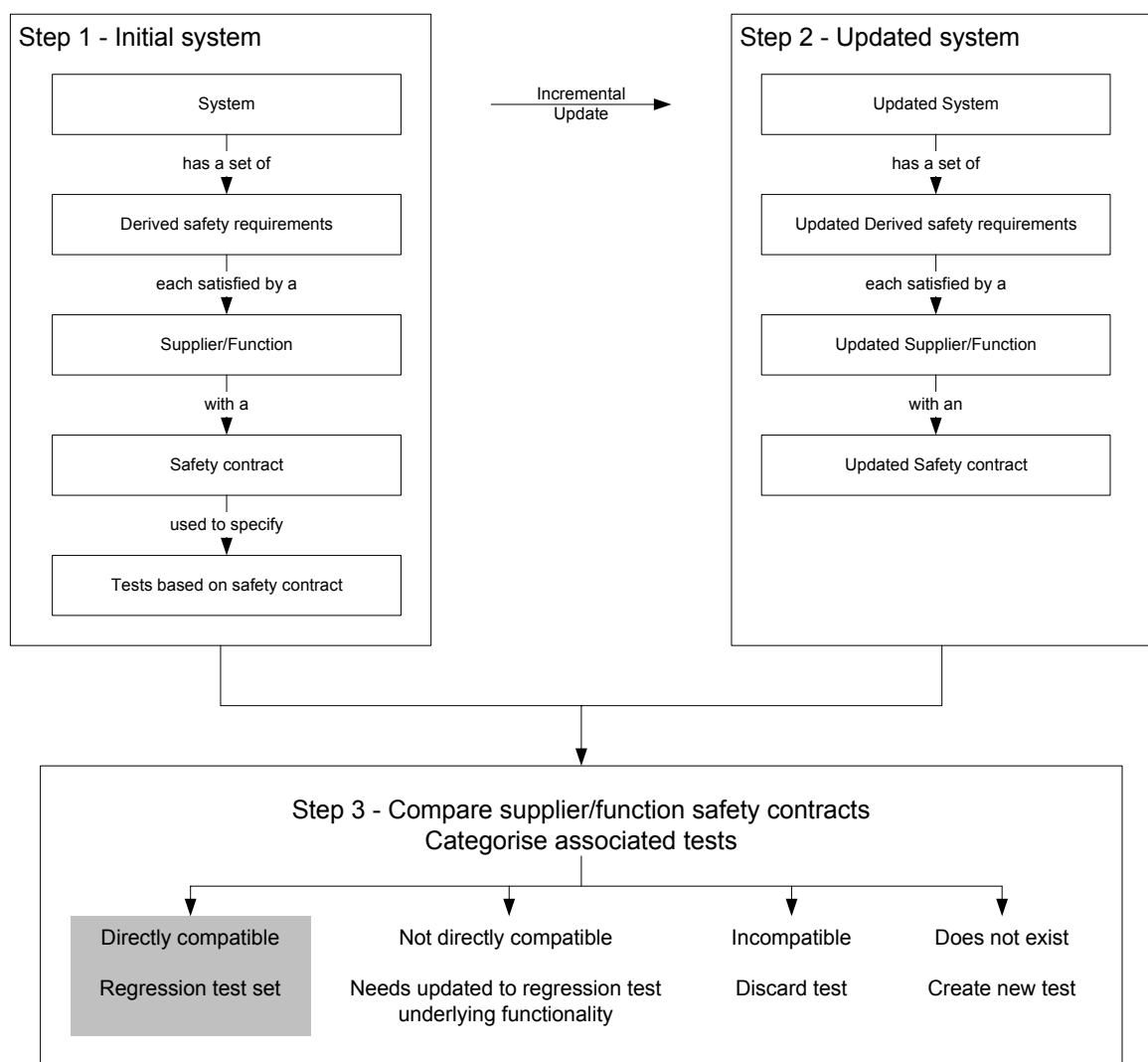


Figure 27 Process to identify regression tests

7.1 Step 1 – Identification of the initial test set

Any system that is to be regression tested must first be tested. If regression tests are to be selected on the grounds of the safety contracts, the Design by Contract rules relating to module exchange, and the work presented in Section 6, then the initial test set must be based on the safety contracts.

Section 2.7 describes the identification of safety contracts within a system. As this technique currently stands, the technique is based on pairs of modules. Therefore, where the satisfaction of a derived safety requirement requires the interaction of more than two modules the functionality needs to be broken down into the constituent pairs of modules.

Section 2.8 covers the development of tests within the context of Design by Contract, and (by extrapolation) safety contracts.

At the end of the test development process:

- the derived safety requirements will have been identified and consolidated,
- the Supplier and Supplier functionality that will satisfy the derived safety requirements will have been identified, along with the Clients that will use the functionality, as Client/Supplier/Functionality triplets,
- the safety contract will have been defined for each of the Client/Supplier/Functionality triplets, and
- tests will have been developed, with the developed tests being based on the safety contracts.

Once the tests have been developed, the tests can be run, with correction as necessary (within either the system or the test set, as appropriate to the location of the failure found), until a point is reached where the tests show that the system is acceptably correct.

7.2 Step 2 – Identification of the updated safety contracts

As a system is updated, the system needs to be re-analysed to identify the updated safety contracts. At the end of the re-analysis process:

- the derived safety requirements will have been updated and re-consolidated, with additional derived safety requirements being added or unnecessary derived safety contracts being removed as the system changes dictates,
- the Supplier and Supplier functionality that will satisfy the updated derived safety requirements will have been identified, along with the Clients that will use the functionality, as updated Client/Supplier/Functionality triplets, and
- the safety contract will have been re-defined for each of the Client/Supplier/Functionality triplets.

Clearly, the amount of work involved with this step will vary greatly, depending on the nature of the changes. Where a like for like module substitution is being carried out, or a localised change within a module is being made (e.g. a bug fix), then there will be little or no change to the safety contracts, and a substantial part of the initial analysis can be re-used. As the changes within the system become greater, the re-analysis will also take longer, since there will be more changes within the derived safety requirements, the Client/Supplier pairs will differ, and as a result the safety contracts will show more variance against the safety contracts derived in Step 1.

7.3 Step 3 – Compare safety contracts to identify regression tests

Having carried out Step 1 and Step 2, Client/Supplier/Functionality triplets and their safety contracts will have been identified for the original system, and for the updated system. The next step is to compare the before update and after update Client/Supplier/Functionality triplets and their safety contracts, using the rules identified in Figure 26.

For each of the updated Client/Supplier/Functionality triplet, the aim is to identify the equivalent original Client/Supplier/Functionality triplet. The safety contracts associated with the original and updated triplet should then be compared.

7.3.1 Safety contract follows rules – directly compatible regression tests

Where the **updates to the safety contract follow the rules identified in Figure 26**, the tests associated with the Client/Supplier/Functionality triplet continue to hold, are **directly compatible**, and **form a part of the regression test set**.

For example, consider the Supplier safety contract introduced in Section 5.1. **Initially**, the Supplier safety contract was:

```
Pre contract
  Pre-condition
    -180 <= steering_wheel_angle <= 180 [degrees]
    vehicle_speed >= 1 [km/hour]
    steering_flag = 1 [state definition]
    ...
  Rely conditions
    10 <= battery_voltage <= 14 [Volts]
    ...
Post contract
  Post-condition
    0 <= command <= 20 [power to be supplied]
    (steering_state = 0) or( steering_state = 1) or (steering_state = 2) [state
    definition]
    ...
  Guarantee conditions
    0 <= actuator_rate_of_change <= 10 [degrees per second]
    ...
```

Following update, the Supplier safety contract may become:

```
Pre contract
  Pre-condition
    -360 <= steering_wheel_angle <= 360 [degrees]
    vehicle_speed >= 1 [km/hour]
    steering_flag = 1 [state definition]
    ...
  Rely conditions
    10 <= battery_voltage <= 14 [Volts]
    ...
Post contract
  Post-condition
    0 <= command <= 20 [power to be supplied]
    (steering_state = 0) or( steering_state = 1) or (steering_state = 2) [state
    definition]
    ...
  Guarantee conditions
    0 <= actuator_rate_of_change <= 5 [degrees per second]
    ...
```

This change results in a weaker pre-contract, and a stronger post-contract. Using the rules presented in Section 6.10, and as highlighted in Figure 28, this is a change where the initial test will continue to be valid, and so form part of the regression test set.

Nature of change to Supplier function	No change to pre-contract	Weaker pre-contract	Stronger pre-contract	Complex change to pre-contract
No change to post-contract	Yes	Yes	No	No
Weaker post-contract	No	No	No	No
Stronger post-contract	Yes	Yes	No	No
Complex change to post-contract	No	No	No	No

Figure 28 Regression test selection rules applied to weaker pre-contract/stronger post-contract

7.3.2 Safety contract does not follow rules – not directly compatible tests

Where the **updates to the safety contract do not follow the rules** identified in Figure 26, the initial tests associated with the Client/Supplier/Functionality triplet do not continue to hold. However, the underlying functionality required by a specified Client continues to be provided by a specified Supplier. As a result this underlying functionality may be regression tested. However, the associated initial tests are **not directly compatible**, and need to be updated to reflect the changes to the safety contracts.

For example, consider the Supplier safety contract described in Section 7.3.1. **Following update**, the Supplier safety contract may become:

```

Pre contract
  Pre-condition
    -100 <= steering_wheel_angle <= 100 [degrees]
    vehicle_speed >= 1 [km/hour]
    steering_flag = 1 [state definition]
    ...
  Rely conditions
    10 <= battery_voltage <= 14 [Volts]
    ...
Post contract
  Post-condition
    0 <= command <= 20 [power to be supplied]
    (steering_state = 0) or (steering_state = 1) or (steering_state = 2) [state definition]
    ...
  Guarantee conditions
    0 <= actuator_rate_of_change <= 15 [degrees per second]
    ...

```

This change results in a stronger pre-contract, and a weaker post-contract. Using the rules presented in Section 6.10, and as highlighted in Figure 29, this is a change where the initial test will not continue to be valid, and so will not form part of the regression test set.

Nature of change to Supplier function	No change to pre-contract	Weaker pre-contract	Stronger pre-contract	Complex change to pre-contract
No change to post-contract	Yes	Yes	No	No
Weaker post-contract	No	No	No	No
Stronger post-contract	Yes	Yes	No	No
Complex change to post-contract	No	No	No	No

Figure 29 Regression test selection rules applied to weaker pre-contract/stronger post-contract

However, it may be that the Client can supply values that will satisfy the updated pre contract clause placed on the value of steering_wheel_angle, and the system is able to cope with post contract clause relating to actuator_rate_of_change. As a result, it would be valid to update the test, and then test the

updated functionality. This type of evaluation would need to be carried out on a case-by-case basis, for each Client/Supplier/Functionality triplet where the safety contract was not directly compatible. In essence:

- the functionality from the original system needs to remain in the updated system, and
- the safety contract in the updated system is not compatible with the safety contract in the original system,

but

- the Client and system can be shown to comply with the updated pre-contract and accept the updated post-contract.

Tests updated because they are not directly compatible would not form part of the regression test set for the current round of testing, since they must be updated to make the test valid again. The tests may become part of the regression test set in subsequent rounds of testing, if the rules developed in Section 6 apply in the subsequent rounds of testing.

7.3.3 Safety contract does not follow rules – other outcomes

Client/Supplier/Functionality triplets that were valid in the original system may be no longer valid in the updated system. The tests associated with these Client/Supplier/Functionality triplets are no longer of interest, and the **tests can be discarded from the current test set**, although these tests should (of course) be maintained within the configuration management system, since the discarded tests form part of the test audit trail.

Client/Supplier/Functionality triplets that did not exist for the original system may have been found within the new system. **New tests**, which do not form part of the regression test set, will be required to cover these Client/Supplier/Functionality triplets.

8 GSN Patterns

Section 6 has identified a set of rules, which can be used to identify regression tests in updated modular systems, where the testing is based on derived safety requirements and safety contracts. A process for applying these rules is identified in Section 7.

This section develops goal structured notation patterns and a module to support safety case arguments for:

- The GSN in Section 8.1 supports generic regression testing. This pattern is high level, encapsulating a general process by which regression testing may be carried out.
- The GSN in Section 8.2 supports the identification of system derived safety requirements and safety contracts. This has been identified as a module, because it was realised that the GSN could be used in a number of situations. This module encapsulates a process by which the derived safety requirements and safety contracts for a system may be identified.
- The GSN in Section 8.3 supports regression testing based on a system's derived safety requirements and safety contracts. The generic regression testing pattern is extended, by basing the testing and regression test selection on derived safety requirements and safety contracts using the rules identified in Section 6, using the process identified in Section 7. This pattern brings together the generic regression testing pattern from Section 8.1 and the identification of system derived safety requirements and safety contracts module from Section 8.2.

8.1 Generic GSN pattern for regression testing

This section describes the GSN pattern shown in Figure 30 Generic Regression Testing.

Pattern Name: Generic Regression Testing Pattern		
Author: C Hollinshead		
Created: 8 th June 2008		Last Modified: 29 th June 2008
Intent		
This pattern provides a framework for arguing that system safety functionality has not regressed following update of a system. The means of demonstrating this is to carry out regression testing, reusing tests from the initial cycle of testing within the testing cycle that follows system update. However, the method by which the regression tests are identified is not specified within this pattern.		
Also known as		
Motivation		
This pattern was developed: As a generic instantiation of a Regression Testing Pattern, where a limited level of detail has been identified, covering the basic steps within a regression testing cycle. Significant development will be required, to cover specific circumstances.		
Structure		
See Figure 30 Generic Regression Testing		
Participants		
G1	Defines the overall objective of the pattern	
C1	This context should be instantiated to refer to a source that describes the expected system usage	
C2	This context should be instantiated to refer to a source that describes the system updates implemented in changing the system from the original system into the updated system	
S1	Strategy highlights that the argument is being constructed by regression testing the updated system, using the tests used to test the original system	
J1	Justification to highlights that original tests should continue to be valid in an updated system, when the underlying functionality has not regressed	

	G2	Defines the objective of establishing the original system testing, where tests have been developed and run, the test results are understood, and the tests and test results have been baselined
	J2	Justification to highlight that the test result on both the initial system and the updated system are necessary to carry out the comparison required to demonstrate that the system has not regressed
	S2	Strategy highlights that the tests will fully cover the original systems' safety functionality
	J4	Instantiation of justification that testing fully tests system safety functionality
	G5	This goal should be developed with the objective of establishing the original test cases - what is to be tested
	G6	This goal should be developed with the objective of establishing the original test set
	S3	Strategy highlights that testing will lead to test results against the original system being generated
	G7	This goal should be developed with the objective of carrying out testing on the original system
	G8	This goal should be developed with the objective of baselining test results, once testing of the original system has been completed. The baselined results will form the basis of the analysis necessary to confirm that the updated system has not regressed
	G3	Defines the objective of establishing the system regression testing, where the regression tests have been identified and run, the test results are understood, and the tests and test results have been baselined
	S4	Strategy highlights that the regression tests will fully pre-existing system safety functionality
	J5	Instantiation of justification that regression testing fully tests pre-existing system safety functionality
	G9	This goal should be developed with the objective of identifying the test cases necessary to regression test the pre-existing system safety functionality
	G10	This goal should be developed with the objective of identifying the original tests that will be used to form the regression test set to be used with the updated system
	S5	Strategy highlights that the regression testing will lead to test results against the updated system being generated
	G11	This goal should be developed with the objective of carrying out testing on the updated system
	G12	This goal should be developed with the objective of baselining the test results, once testing of the updated system has been completed. The baselined results will form the basis of the analysis necessary to confirm that the updated system has not regressed
	G4	This goal should be developed with the objective of comparing the test results from regression tests with the test results obtained during testing of the initial system
	C3	Context to highlight that the initial test results and the regression test results should be comparable (within the limitations of the system)
	C4	This context should be instantiated to define what regression test results are acceptable within the context of the system changes that have been made, the test results that have been obtained, and the level of variance permitted between test results obtained on the initial system and the regression test result obtained on the updated system

	J3	This justification should be instantiated to justify why the regression test results demonstrate that the system safety functionality has not regressed. This will include explaining any outcomes where the test results from the initial system and the updated system are different, but the difference is acceptable, as a result of system limitations (see C3)
Collaborations	G5/G6/G7/G8 and G9/G10/G11/G12 establish the tests and test results for the original system and the updated, regression tested system. In G4, the aim is to compare the test results from the original system and the updated system, justifying any situations where the results may be different, but the regression test result is still valid in demonstrating that the system safety functionality has not regressed.	
Applicability	Within the context of regression testing, this is a very general pattern. As such, it will have broad applicability across a wide range of systems.	
Consequences	After instantiating the pattern, there should be no unresolved justifications, goals and contexts.	
Implementation	The goals for G2 will be met during the initial testing phase. The goals for G3 will be met during the regression testing phase, following system update. The tasks for G4 can be started towards the end of the G3 work, but cannot be finished until the tasks for G3 are complete	
Examples	None known	
Known Uses	None known	
Related Patterns	DSR and Safety Contract Based Regression Testing Pattern – a more specific version of this pattern, shown in Figure 32 DSR and safety contract based regression testing (1 of 2, see also Figure 33) and Figure 33 DSR and safety contract based regression testing (2 of 2, see also Figure 32)	

8.2 GSN module for identification of system DSRs and safety contracts

This section describes the GSN module referred to in Figure 31 Generic module to identify system DSRs and safety contracts

Module Name: Generic Identification of System DSRs and Safety Contracts		
Author: C Hollinshead		
Created: 29 th June 2008		Last Modified: 5 th September 2008
Intent	This module provides a framework for identifying a system's DSRs and safety contracts	
Also known as		
Motivation	This module was developed: As a generic module for identifying a system's DSRs and safety contracts. The need for this module was identified while developing a DSR and Safety Contract Based Regression Testing Pattern (see Section 8.3), where this activity is carried out twice (once on the initial system, and once on the updated system). It is thought that this module may also be reusable in other situations (e.g. to support a goal that all a system's derived safety requirements have been identified, along with their interface specifications, during the requirement specification phase of a project).	
Structure	See Figure 31 Generic module to identify system DSRs and safety contracts	
Participants	G1	Defines the overall objective of the module

	S1	Strategy highlights that the argument is being constructed through a sequence of activities. First the derived safety requirements must be identified. Then the means of delivering the derived safety requirements through interactions between Clients, Suppliers and the Suppliers' functionality must be identified. Finally, the system safety contracts can be established, for each of the Client/Supplier/Functionality triplets
	G2	Defines the objective of establishing the derived safety requirements for the system
	C1	Context to highlight that derived safety requirements defines what is required with respect to safety functionality
	S2	Strategy highlights that G2 is achieved through the application of system safety analysis
	G5	Defines the objective of identifying the system hazards
	J2	Justification to highlights that identified hazards will be removed, reduced and mitigated via the derived safety requirements
	S4	Strategy highlights that G5 is achieved through the application of a well known and understood system safety analysis technique
	G6	This goal should be developed with the objective of defining the system boundary, so that the preliminary hazard analysis can be carried out
	G8	This goal should be developed with the objective of identifying the accidents that can happen as a result of a system failure
	G7	This goal should be instantiated and developed with the objective of identifying the hazards associated with, and leading to, each of the accidents identified in G8
	G9	Defines the objective of identifying the consolidated DSRs which have been identified to prevent, manage and mitigate the identified system hazards
	S5	Strategy highlights that G9 is achieved through the application of a well known and understood system safety analysis technique
	G10	Defines the objective of identifying the components within the system boundary
	G11	This goal should be developed with the objective of describing the system architecture
	G12	This goal should be developed with the objective of identifying the components within the system architecture
	G13	This goal should be instantiated and developed with the objective of identifying the role that each components within the system architecture must perform
	G14	Defines the objective of identifying the failure modes associated with the system components within the system boundary
	G15	This goal should be instantiated and developed with the objective of identifying the failure modes of each of the identified components, where the failure mode could contribute to a hazardous situation arising
	G16	This goal should be instantiated and developed with the objective of identifying the impacts and mitigations that can be put in place to manage each of the identified failure modes
	G17	This goal should be instantiated and developed with the objective of identifying the derived safety requirements that will manage and mitigate the failure modes identified for each of the identified components
	G18	This goal should be instantiated and developed with the objective of identifying the core derived safety requirements, with duplicates derived safety contracts removed
	J3	Justification to highlights that consolidation of the derived safety requirements will remove duplicate derived safety requirements

	G3	Defines the objective of identifying the component inter-relationships. In effect, this is a derivation of the Client/Supplier pairs that exist within the system. It is these pairs, where the Client is able to call functionality available within the Supplier, that are used to satisfy the derived safety requirements
	J1	Justification to highlights that the Client/Supplier interrelationships are key to supplying the functionality necessary to satisfy the derived safety requirements
	S3	Strategy highlights that G3 is achieved through the application of a well known and understood system safety analysis technique
	G19	This goal should be instantiated and developed with the objective of identifying the Supplier and Supplier functionality that will satisfy each of the derived safety requirements
	G20	This goal should be instantiated and developed with the objective of identifying the Clients that will make use of the Suppliers and Supplier functionality that will satisfy each of the derived safety requirements
	G4	Defines the objective of identifying the safety contract associated with each of the identified Client/Supplier/Functionality triplet
	C2	Context to highlight that safety contracts defines the operational limitations of and Client/Supplier/Functionality triplet, and so specify the limitations where by a Client/Supplier/Functionality triplet can satisfy a derived safety requirement
	G21	This goal should be instantiated and developed with the objective of defining the functional component of each of the identified Client/Supplier/Functionality triplet
	G22	This goal should be instantiated and developed with the objective of defining the architectural component of each of the identified Client/Supplier/Functionality triplet
	G23	This goal should be instantiated and developed with the objective of defining the behavioural component of each of the identified Client/Supplier/Functionality triplet
	G24	This goal should be instantiated and developed with the objective of defining the quantifiable component of each of the identified Client/Supplier/Functionality triplet
Collaborations		The sub-goals G5 and G9 combine to generate a set of consolidated derived safety requirements, as described in G2. This part of the module has the potential to exist as a module in its own right (e.g. to support an argument that a system's safety requirements have been identified during the system safety requirements specification phase of a project). The sub-goals G2 and G3 combine to generate the Client/Supplier/Functionality triplets that will use and deliver the derived safety requirements.
Applicability		This module is specific to a system where the interactions have been reduced to a single Client interacting with a single Server, without any consideration being given the way in which a Client call may cascade beyond the Supplier to which the call is made.
Consequences		After instantiating the module, there should be no unresolved justifications, goals and contexts.
Implementation		This safety case module is an encapsulation of the process that needs to be followed to arrive at the safety contracts within a system. As a result, the sub-goals of G2 need to fulfilled, before the sub-goals of G3 can be completed. Likewise, the sub-goals of G3 need to fulfilled, before the sub-goals of G4 can be completed. The sub-goals of G2 could be developed in isolation. However, the sub-goals of G3 and sub-goals of G4 are not 'stand alone - there is a sequential dependence, building from G2 to G3, and then on to G4. The sub-goals of G2 will establish the argument for the identification of a system's derived safety requirements.
Examples		None known
Known Uses		None known

Related Modules	None known
Related Patterns	DSR and Safety Contract Based Regression Testing Pattern – this pattern is used as the basis of identifying the parameters to be tested, and the boundaries of testing, as shown in Figure 32 DSR and safety contract based regression testing (1 of 2, see also Figure 33) and Figure 33 DSR and safety contract based regression testing (2 of 2, see also Figure 32)

8.3 GSN pattern for regression testing via DSRs and safety contracts

This section describes the GSN pattern shown in Figure 32 DSR and safety contract based regression testing (1 of 2, see also Figure 33) and Figure 33 DSR and safety contract based regression testing (2 of 2, see also Figure 32)

Pattern Name: DSR and Safety Contract Based Regression Testing Pattern		
Author: C Hollinshead		
Created: 8 th June 2008		Last Modified: 5 th September 2008
Intent	This pattern provides a framework for arguing that system safety functionality has not regressed following update of a system. The argument is based on the use of initial tests and regression tests identified through the system's derived safety requirements, with testing boundaries defined by safety contracts.	
Also known as		
Motivation	This pattern was developed: As a more specific/specialised instantiation of the Generic Regression Testing Pattern, where a greater level of detail has been identified. To test the Generic Regression Testing Pattern under more specific circumstances.	
Structure	See Figure 32 DSR and safety contract based regression testing (1 of 2, see also Figure 33) and Figure 33 DSR and safety contract based regression testing (2 of 2, see also Figure 32)	
Participants	G1	Defines the overall objective of the pattern
	C1	This context should be instantiated to refer to a source that describes the expected system usage
	C2	This context should be instantiated to refer to a source that describes the system updates implemented in changing the system from the original system into the updated system
	S1	Strategy highlights that the argument is being constructed by regression testing the updated system, using the tests used to test the original system
	J1	Justification to highlights that original tests should continue to be valid in an updated system, when the underlying functionality has not regressed
	G2	Defines the objective of establishing the original system testing, where tests have been developed and run, the test results are understood, and the tests and test results have been baselined
	J2	Justification to highlight that the test result on both the initial system and the updated system are necessary to carry out the comparison required to demonstrate that the system has not regressed
	S2	Strategy highlights that the tests will fully cover the original systems' safety functionality
	J4	Justification that DSRs are fully identified
	J5	Justification that safety contracts define boundaries of operation for the functionality implemented to satisfy the DSRs

	J6	Justification that the tests will be based on the functionality to satisfy the DSRs within the scope of the safety contract boundaries
	G5	This goal defines the objective of establishing the original test cases – what is to be tested. This is achieved through the use of the Generic Identification of System DSRs and Safety Contracts module
	G6	This goal defines the objective of establishing the original test set
	S6	This argument should be developed to support the intent of developing the test set using the DSRs and the safety contracts as the basis for this development
	J8	Justification to highlight that the DSRs define what is to be tested
	J9	Justification to highlight that the safety contracts define the operational limits for the DSRs
	S3	Strategy highlights that testing will lead to test results against the original system being generated
	G7	This goal should be developed with the objective of carrying out testing on the original system
	G8	This goal should be developed with the objective of baselining test results, once testing of the original system has been completed. The baselined results will form the basis of the analysis necessary to confirm that the updated system has not regressed
	G3	Defines the objective of establishing the system regression testing, where the regression tests have been identified and run, the test results are understood, and the tests and test results have been baselined
	S4	Strategy highlights that the regression tests will fully pre-existing system safety functionality
	J7	Instantiation of justification that regression testing fully tests pre-existing system safety functionality
	G9	This goal defines the objective of establishing the updated test cases – what is to be tested. This is achieved through the use of the Generic Identification of System DSRs and Safety Contracts module
	G10	Defines the objective of identifying the original tests that will be used to form the regression test set to be used with the updated system
	S7	Strategy highlights that the regression test set will be identified by using the substitution rules embedded within Design by Contract
	C5	Context to highlight that the substitution rules within Design by Contract can be used to select regression tests from the initial test set
	A1	Assumption to highlight that this strategy assumes that the system has been reduced to its component Client/Server pairs
	G13	This goal should be developed with the objective of identifying the Client/Server/Functionality triplets that exist in both the original system and the updated system
	G14	Defines the objective of carrying out pair wise comparisons of the safety contracts associated with the triplets identified in G13
	G15	This goal should be developed with the objective of identifying the regression test set, from Client/Server/Functionality triplets where the updated safety contract is compatible with the initial safety contract
	Sn1	The set of regression tests compatible with the updated Client/Server/Functionality triplets
	C6	Context to highlight that the Client/Server/Functionality triplets and safety contracts are compatible within the context of Design by Contract modules substitution rules

	G16	This goal should be developed with the objective of identifying the test set where the underlying functionality still exists, but the updated safety contracts are not compatible with the initial safety contract. These will be identified by Client/Server/Functionality triplets that exist for both the initial system, and the updated system, where the safety contracts do not follow the identified rules.
	G17	This goal should be developed with the objective of identifying the tests associated with functionality that is no longer used in the updated system. These will be identified by Client/Server/Functionality triplets that exist for the initial system, but are not identified within the updated system
	G18	This goal should be developed with the objective of identifying the test cases that will be needed to support testing of the updated system. These will be identified by Client/Server/Functionality triplets that do not exist for the initial system, but are identified within the updated system
	Sn2	The set of initial tests that are not compatible with the updated Client/Server/Functionality triplets
	C7	Context to highlight that the Client/Server/Functionality triplets and safety contracts are not compatible within the context of Design by Contract modules substitution rules
	S5	Strategy highlights that the regression testing will lead to test results against the updated system being generated
	G11	This goal should be developed with the objective of carrying out testing on the updated system
	G12	This goal should be developed with the objective of baselining the test results, once testing of the updated system has been completed. The baselined results will form the basis of the analysis necessary to confirm that the updated system has not regressed
	G4	This goal should be developed with the objective of comparing the test results from regression tests with the test results obtained during testing of the initial system
	C3	Context to highlight that the initial test results and the regression test results should be comparable (within the limitations of the system)
	C4	This context should be instantiated to define what regression test results are acceptable within the context of the system changes that have been made, the test results that have been obtained, and the level of variance permitted between test results obtained on the initial system and the regression test result obtained on the updated system
	J3	This justification should be instantiated to justify why the regression test results demonstrate that the system safety functionality has not regressed. This will include explaining any outcomes where the test results from the initial system and the updated system are different, but the difference is acceptable, as a result of system limitations (see C3)
Collaborations		G5/G6/G7/G8 and G9/G10/G11/G12 are sequences of goals that establish the tests and test results for the original system and the updated, regression tested system. G5/G6 and G9/G10 develop the test and regression test sets. G7/G8 and G11/G12 develop the test results for the original and updated system. In G4, the aim is to compare the test results from the original system and the updated system, justifying any situations where the results may be different, but the regression test result is still valid in demonstrating that the system safety functionality has not regressed.

Applicability	<p>Within the context of regression testing based on integer safety contracts, this is a very general pattern. As such, it will have broad applicability across a wide range of systems.</p> <p>The starting point for this pattern is a well characterised system of sub-systems, where the interactions between the sub-systems are well understood. This understanding is necessary to establish the Client/Supplier/Functionality triplets that deliver the safety functionality in support of the derived safety requirements.</p> <p>In addition to the Client/Supplier/Functionality triplets, it is also necessary to establish the safety contracts. This requires a good understanding of the variable values that are sent from and may be received by the sub-systems within the system.</p>
Consequences	After instantiating the pattern, there should be no unresolved justifications, goals and contexts.
Implementation	<p>The goals for G2 will be met during the initial testing phase.</p> <p>The goals for G3 will be met during the regression testing phase, following system update.</p> <p>The tasks for G4 can be started towards the end of the G3 work, but cannot be finished until the tasks for G3 are complete</p>
Examples	None known
Known Uses	None known
Related Patterns	Generic Regression Testing Pattern – a more generic version of this pattern shown in Figure 30 Generic Regression Testing.

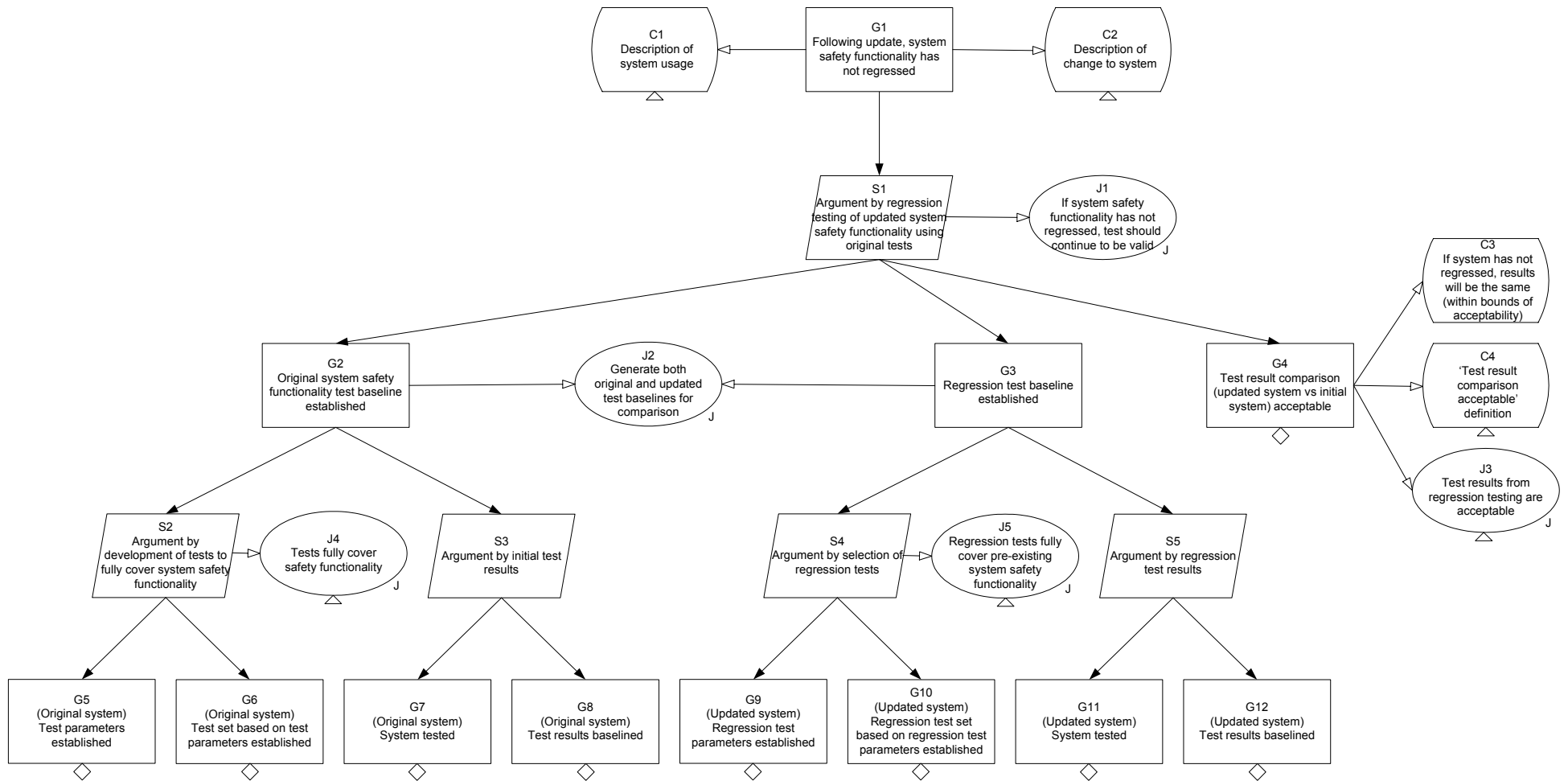


Figure 30 Generic Regression Testing

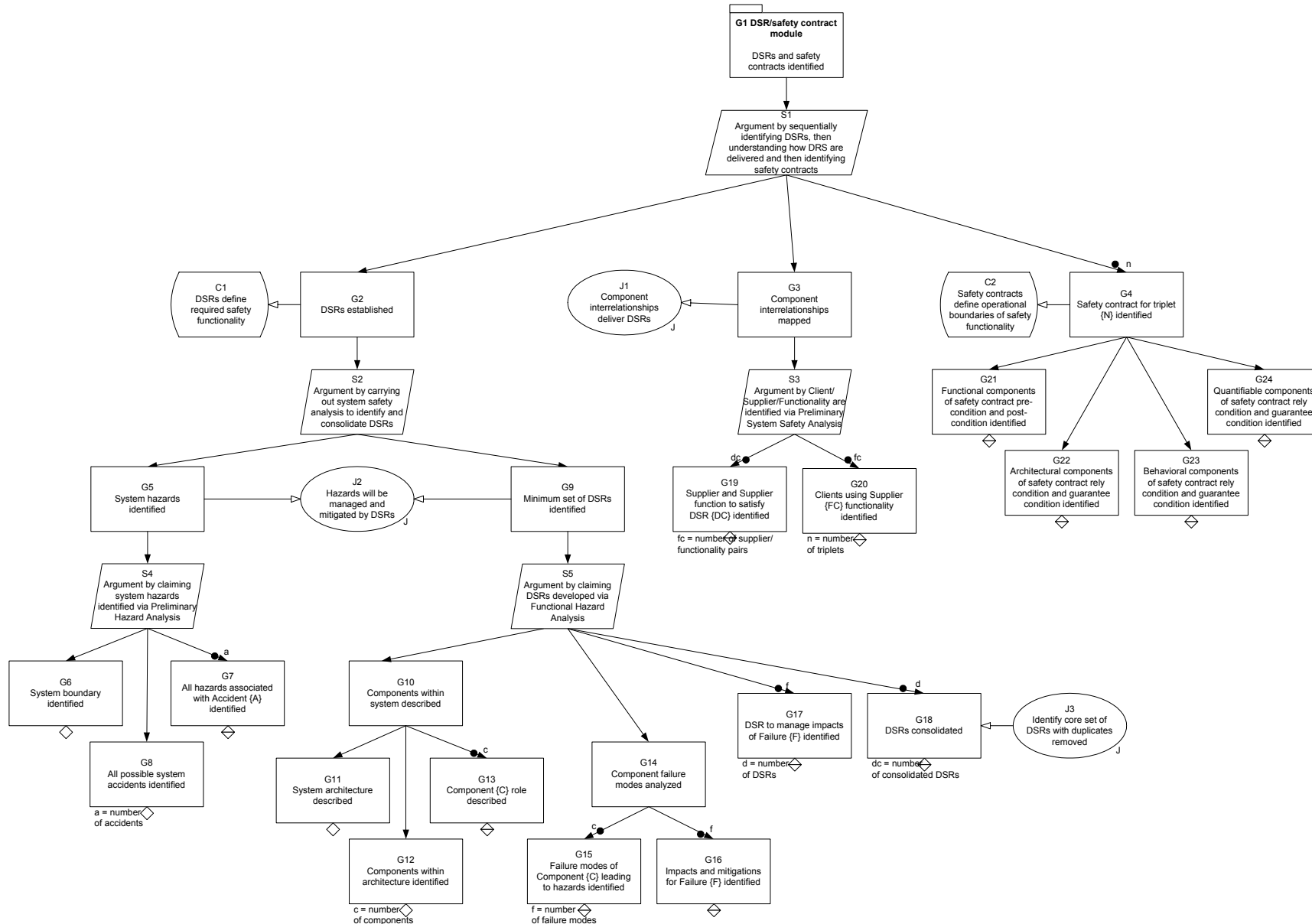


Figure 31 Generic module to identify system DSRs and safety contracts

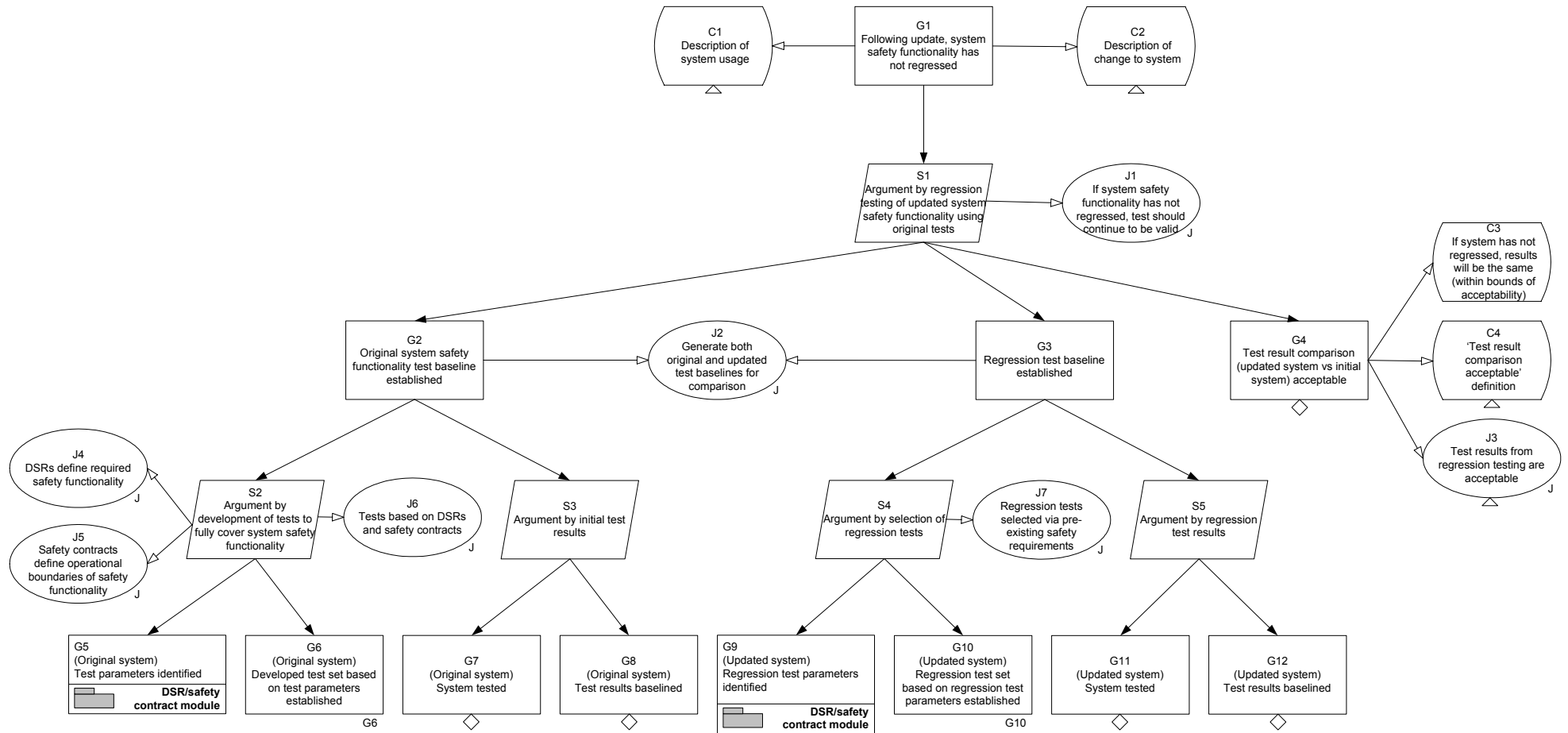


Figure 32 DSR and safety contract based regression testing (1 of 2, see also Figure 33)

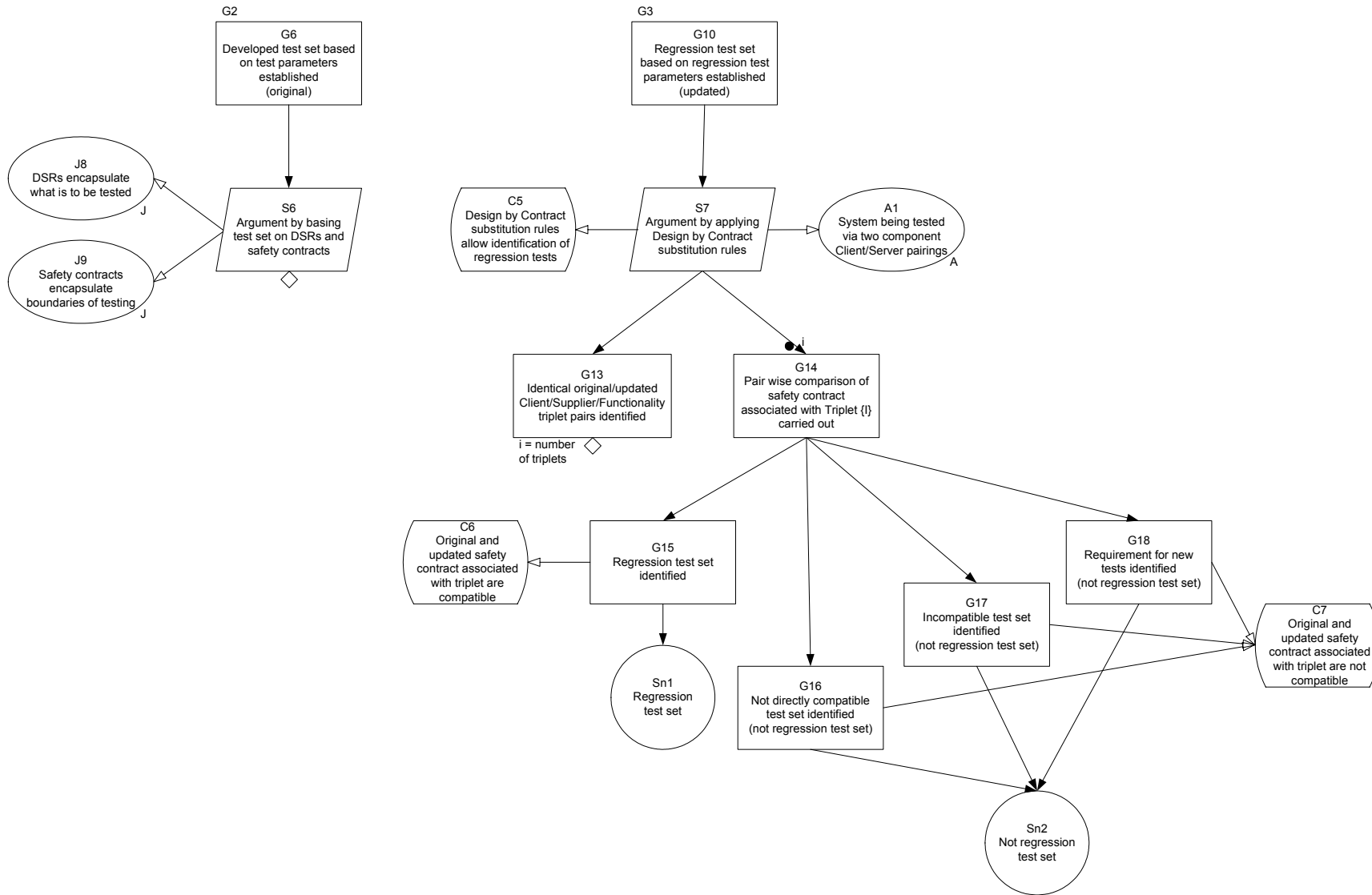


Figure 33 DSR and safety contract based regression testing (2 of 2, see also Figure 32)

9 Integer non-functional safety contracts and regression testing

Section 6 covered the use of integer based functional safety contracts, and safety contract based testing, for the identification of regression tests in a simple, two component system where the Supplier has been updated. The work in Section 6 was tightly constrained, to reduce the problem to a potentially solvable one.

This section seeks to remove one of the constraints placed on the initial problem, and so broaden the nature of the safety contracts under consideration. This section moves from integer, functional safety contracts to integer, non-functional safety contracts, in a simple, two component system where the Supplier has been updated.

9.1 A typical safety critical system – non-functional requirements

Within any system development project, the focus (naturally) tends to be on the functionality that the system will offer. However, to support the required functionality other, non-functional requirements, will come into play. Typical non-functional requirements relate to:

- timing constraints,
- usability,
- safety integrity levels (SILs), and
- reliability, availability and maintainability (RAM),

Section 9.2 to 9.3 will focus on timing constraints.

Section 9.4 to 9.5 will focus on SIL constraints.

9.2 Timing constraints

Timing in this context refers to the length of time required to execute a defined piece of functionality.

Generally, execution time needs to:

- be below a certain value (e.g. 10ms),
- happen at a certain point in the programme sequence (e.g. as the first item in a routine), and/or
- happen at a certain point in the processor cycle (e.g. as an interleaved task, where the cycle time is 20ms, and the ongoing background processing takes 15 ms, the interleaved task would have to occur in the last 5 ms of the cycle).

The focus here is on the time taken for a Supplier to execute a unit of Client called functionality.

In particular, the ECU within the active steering system operates on a 20ms cycle. Within each cycle, a range of tasks must be completed (e.g. processing of sensor input, checking of memory, construction and passing of messages to CAN). As a result, the core functionality of driving the actuator within the active steering system must occur within a time limited period within each cycle.

As in Scenario 1, the Client is the system control software in the ECU. The Supplier is the actuator control software. The scenario is shown diagrammatically in Figure 34.

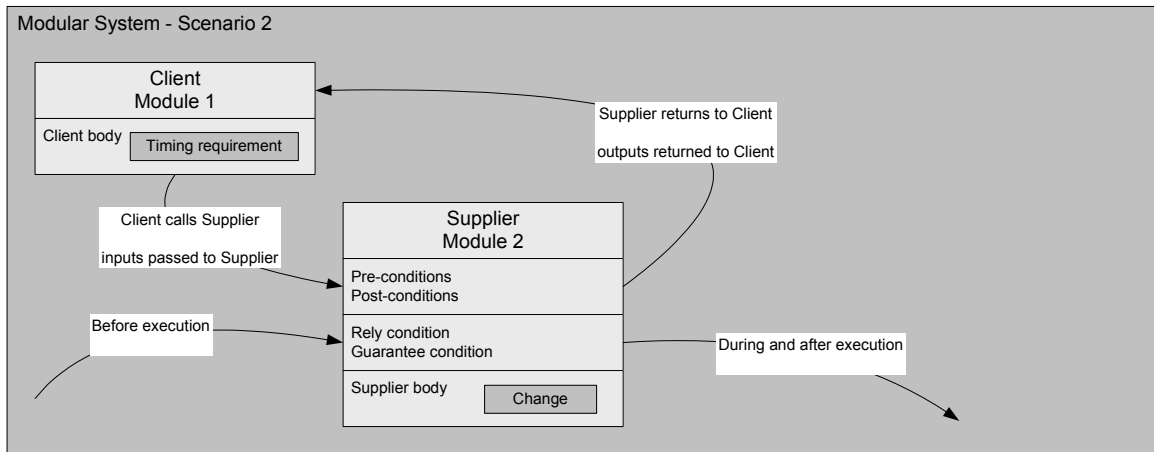


Figure 34 Scenario 2 – A two module Client/Supplier system with a timing restriction

Generally, it is the Client that will be concerned with the time it takes to execute a unit of functionality. Where a number of tasks have to be achieved within a limited time frame (e.g. as described above), the total time available will be allocated between the tasks. Typically, this is done during the requirements analysis and design phases – the system’s analysis will allocate timings to the sub-systems, with the sub-systems further breaking the timings down to the sub-sub-systems. Sanity checking then occurs during implementation, with timings being tweaked, as the true picture emerges.

As a result, a Client will place restrictions on one or more of its Suppliers. This is the inverse of Scenario 1, where the Supplier specifies what it required to execute correctly, and what the Supplier will ensure. In this instance, the Client needs to ensure that the Supplier acts in a specific way, if the Client is going to deliver on its own functionality.

9.3 Scenario 2 – timing constraints and safety contracts

How do timing constraints fit within the framework of a safety contract, and regression testing? There are two possible approaches:

- Approach 1: Supplier will supply, and
- Approach 2: Client ‘depends-on’ Supplier.

9.3.1 Approach 1: Supplier will supply

The Suppliers could define the time that it will take to deliver a piece of functionality within the rely condition – the Supplier is reliant on the processor being able to offer an identified amount of processing time, if the Supplier is to deliver its functionality before another process is started via an interrupt.

In addition to the integer-based clauses within the safety contract first described in Section 5.1, an additional clause would be identified, relating to timing:

```

Supplier safety contract for the function COMMAND
Supplier safety contract
Pre-condition
-180 <= steering_wheel_angle <= 180 [degrees]
vehicle_speed >= 1 [km/hour]
steering_flag = 1 [state definition]
...
Rely conditions
10 <= battery_voltage <= 14 [Volts]
processor_time_available >= 5 [milli seconds]
...
Post-condition

```

```
0 <= command <= 20 [power to be supplied]
(steering_state = 0) or( steering_state = 1) or (steering_state = 2)
[state definition]
```

...

Guarantee conditions

```
0 <= actuator_rate_of_change <= 10 [degrees per second]
```

...

However, timing is dependent on a number of things, including processor speed and network speed. As a result, this type of rely condition needs to be identified on a system by system basis, with the assumptions upon which the rely condition was predicated being made explicit.

As a rely condition, it would form part of the safety contract, and follow the rules discussed in Section 6.

But, even if it is the Supplier that must deliver the timing, timing is a property being driven by the Client. This leads to the second approach.

9.3.2 Approach 2: Client 'depends-on' Supplier

As discussed above, timing is a property dictated by the Client, and dependent on the system within which both the Client and Supplier operate. This could be described within an inverse relationship, where the Client '**depends-on**' a Supplier to deliver within a specified timeframe. Again, the assumptions upon which the depends-on condition was predicated need to be made explicit.

While the Client cannot influence the Supplier and/or system with respect to delivery of conditions contained in the depend-on specification (other than by fulfilling the pre-contract, so ensuring that the Supplier can deliver its promised functionality), defining the depends-on relationship does make any dependencies of this type explicit. It offers a way of providing documentation within the system, in keeping with the use of pre-conditions and post-conditions within Design by Contract to document software boundaries.

In addition to the integer-based clauses within the safety contract first described in Section 5.1, an additional clause would be identified, relating to timing:

Client safety contract for the command function

Client depends-on relationship

```
processing_time >= 5 [milli seconds]
```

Supplier safety contract for the function COMMAND

Supplier safety contract

Pre-condition

```
-180 <= steering_wheel_angle <= 180 [degrees]
```

```
vehicle_speed >= 1 [km/hour]
```

```
steering_flag = 1 [state definition]
```

...

Rely conditions

```
10 <= battery_voltage <= 14 [Volts]
```

...

Post-condition

```
0 <= command <= 20 [power to be supplied]
```

```
(steering_state = 0) or( steering_state = 1) or (steering_state = 2)
[state definition]
```

...

Guarantee conditions

```
0 <= actuator_rate_of_change <= 10 [degrees per second]
```

...

As with defining a safety contract, and then using the safety contract to define test conditions, the depends-on relationship also has the potential to be used to define test conditions. One way of approaching this would be to ‘transfer’ the depends-on relationship into the safety contract (i.e. place the Client’s depends-on conditions within the relevant Supplier safety contract) at the time the modules are brought together within a system. This allows flagging of the Client’s part of a safety contract in the place where the need is being driven from. It also allows testing to show delivery of the Clients safety contract in the location where the delivery is actually being made. Using this approach, the timing would (for testing purposes) form part of the safety contract, and follow the rules discussed in Section 6.

9.4 SIL constraints

A module’s SIL requirements dependent on:

- the overall SIL of the system, as determined through analysis following the standards being referenced, and
- the way in which SILs have been apportioned, depending on the standards being referenced, whether independent x out of y voting is being used, whether the module is the only item providing functionality, or whether monitoring functions are being applied, etc.

This work will focus on the SIL requirements for a module, but will not concern itself with how the SIL value of a module is derived, or the actual validity of SILs as a means to measure the potential safety of a system’s components.

For this type of system, following BS EN 61508 [55], the functionality of the working system is typically classified at SIL 3. Both the Client and the Supplier in this example need to be rated as SIL 3. As in Scenario 1, the Client is the system control software in the ECU. The Supplier is the actuator control software. The scenario is show diagrammatically in Figure 35.

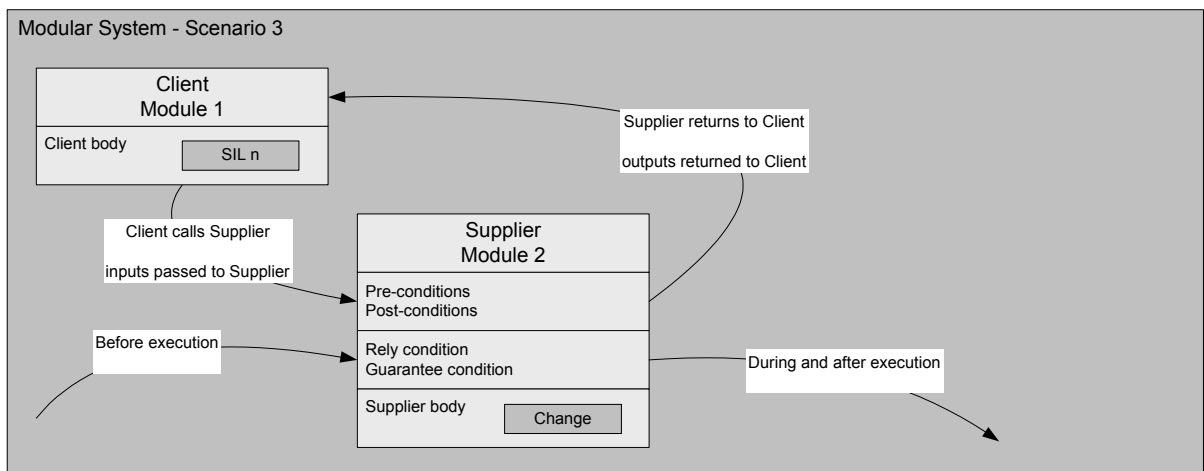


Figure 35 Scenario 3 – A two module Client/Supplier system with a SIL restriction

In this case, it is the system as a whole that must reach a given SIL. The individual components within the system must contribute to the system SIL by being of specified (via system analysis) SILs. Likewise, any work carried out to interface the components must also sit within the context of the overall SIL of the system.

Unlike functional integer safety contracts and timing constraints that can be tested for, SILs are built into a module through process and design. As a result, SILs cannot be tested in the conventional way, by exercising the system. Instead, the SIL of a component needs to be demonstrated. There are two approaches to this. Either:

- The component’s developer needs to demonstrate to integrators that a quoted SIL is correct. However, historically, component developers have been reluctant to release design and development

process information to their product's integrators. The developers want to sell 'black boxes', in the truest meaning of the phrase.

OR

- A component is certified via some third party (e.g. BSI, TÜV), so providing a neutral interface between developer and integrator.

9.5 Scenario 3 – SIL constraints and safety contracts

How do SIL constraints fit within the framework of a safety contract, and regression testing? As with timing constraints, there are two possible approaches:

- Approach 1: Supplier will supply, and
- Approach 2: Client 'depends-on' Supplier.

9.5.1 Approach 1: Supplier will supply

The Supplier could define the SIL that it has been developed to. This could be specified using the invariant clauses defined within Design by Contract. This adds an extra clause within the safety contract:

```
Supplier safety contract for the function COMMAND
Supplier safety contract
  Pre-condition
    -180 <= steering_wheel_angle <= 180 [degrees]
    vehicle_speed >= 1 [km/hour]
    steering_flag = 1 [state definition]
    ...
  Rely conditions
    10 <= battery_voltage <= 14 [Volts]
    processor_time_available >= 5 [milli seconds]
    ...
  Post-condition
    0 <= command <= 20 [power to be supplied]
    (steering_state = 0) or (steering_state = 1) or (steering_state = 2)
    [state definition]
    ...
  Guarantee conditions
    0 <= actuator_rate_of_change <= 10 [degrees per second]
    ...
Invariant
  SIL 3
  ...
```

Invariant is a suitable place to put this type of property, since invariants are used to specify properties that do not change over the course of execution, within a particular release of a module. As discussed, unlike timing constraints, the demonstration will be by examination or certification, rather than testing of functionality.

9.5.2 Approach 2: Client 'depends-on' Supplier

As discussed above, SIL is a property dictated by the Client, and will depend upon the system type, and the integrity level that governs the system type. This could be described within an inverse relationship, where the Client 'depends on' a Supplier to deliver at or above a specified integrity level. In this situation, the identified SIL for the Supplier will be the minimum permissible – it would be acceptable to use a Supplier with a higher SIL.

In addition to the integer-based clauses within the safety contract first described in Section 5.1, an additional clause would be identified, relating to SIL:

Client safety contract for the command function

Client depends-on relationship

SIL >= 3

Supplier safety contract for the function COMMAND

Supplier safety contract

Pre-condition

-180 <= steering_wheel_angle <= 180 [degrees]

vehicle_speed >= 1 [km/hour]

steering_flag = 1 [state definition]

...

Rely conditions

10 <= battery_voltage <= 14 [Volts]

...

Post-condition

0 <= command <= 20 [power to be supplied]

(steering_state = 0) or (steering_state = 1) or (steering_state = 2)

[state definition]

...

Guarantee conditions

0 <= actuator_rate_of_change <= 10 [degrees per second]

...

As previously discussed, unlike timing constrains, the demonstration will be by examination or certification, rather than demonstration of functionality.

10 Evaluation

10.1 Taking this work forwards

This dissertation presents rules and a process for the identification of regression tests in safety critical and safety related modular systems. This dissertation then goes on to present GSN patterns and a GSN module that may be used to defensibly argue that regression testing has been carried out in a complete manner, with a high level of integrity.

The next step is to test the process (and therefore the rules) on a number of real world modular systems. This will serve two purposes:

- Test the process, to see if it is practicable.

Section 10.2 raises a number of issues that may prevent the process from working, or mean that the process is unworkable on grounds of cost, time, resource required, and/or depth of knowledge of the system required. These issues can be checked as the process is applied to real world modular systems.

- Generate the data necessary to populate, and so test, the GSN patterns.

Ideally, the rules and process would initially be trialled on a small, simple modular system, where the system and modules are well understood. If the initial trial proved promising, then the trial should progressively move to larger and less well understood modular systems, as more experience is gained.

Since derived safety requirements are a specialised type of general requirements, a low risk route to testing the process could be to trail it during regression testing on a non-safety critical and non-safety related modular system. While a safety argument would not be needed in this situation, the GSN patterns could still be populated, since the data would be available from testing the process.

A number of suggestions for future work are presented in Section 11.7. In addition to testing the rules, process and GSN patterns (as described above), further work is also required to check the broader applicability of the technique, and undertake the other identified future work.

10.2 Critique

10.2.1 Regression test selection for functional integer safety contracts

The material presented in Section 6 demonstrates that:

- safety contracts, combined with
- the rules relating to component interchange that are embedded within Design by Contract,

can be used to identify regression tests in a simple two component system, where:

- the Client is unchanged, but
- a Supplier function has been updated.

To use this technique for the identification of regression tests (as currently described), it must be applied across the whole system. It is necessary to:

- **Identify all the pairs of Client/Supplier interactions** that contribute to the satisfaction of the derived safety requirements defined for the system. This includes the interactions between a Supplier and its Clients for both the initial version of the Supplier and updated version of the Supplier. However, there may be other interactions between other modules within the system. When a Supplier is called, a

'chain reaction' may be initiated. The Supplier itself may become a Client to other Suppliers. All of these interactions need to be identified.

Where the integrators have developed all the components within a system, identifying all the Client/Supplier interactions will be easier to achieve, since the system should be reasonably well understood. However, even in this situation, there will generally be a level within the system where components have been bought in. Typically this will be at the operating system level for software, and the processor level in hardware.

Where the integrators have not developed all the components, the interactions between components may be less clear. This may make it difficult to ensure that all the potential Client/Supplier interactions within a system have been identified.

One way to address the issue of identifying all the Client/Supplier interactions within the system is to add 'wrappers' to all the components. In this context, the purpose of a wrapper is to remove all the direct interfaces between components. Instead, all interactions with a component are forced to go via the wrapper. This provides a means of controlling the access to each of the components. If an interaction has not been identified, then it will not be supported within a wrapper, and the system will malfunction.

However, wrappers come at a price. Like any other bespoke component, a wrapper needs to be specified, designed, built, tested and maintained. If wrappers are to be used as a means of managing component interactions, the cost to the wrapper needs to be weighed against the benefits that the wrapper may deliver.

- **Identify all the system safety contracts** associated with the functionality within the original versions of the Suppliers, and any updated versions of the system's Suppliers.

Where the system's integrator has developed a Supplier, the Supplier will be well understood. As a consequence of the system and component analysis carried out during development, it should be possible to identify the relevant safety contracts with a high degree of confidence.

Where a Supplier has been obtained from a third party developer, this could be problematic. For example, the third party developer:

- May not be willing to provide the information necessary to identify the safety contracts (e.g. for contractual or liability reasons).
- May not have developed the Supplier with the intent of identifying any safety contracts. As a result, the analysis required to provide the information necessary to identify the safety contract may not have been fully carried out, or may not be carried out in such a way as to provide a high degree of confidence in the outcome.
- May fall foul of one of the problems associated with Design by Contract and the object-oriented paradigm – it is very difficult to anticipate all the circumstances under which a component may be used. As a result, the safety contracts (or component functionality) may not fully fit with the particular circumstances under which a component is used.

Additionally, it is not only the pre-conditions and post-conditions that must be identified. The rely conditions and guarantee conditions must also be identified. This requires an extension of the thinking that goes with Design by Contract, into the wider system. The rely conditions and guarantee conditions are harder to establish – it needs an understanding of how the modules interact at a very fundamental level.

As a result, it may be difficult to ensure that all the safety contracts within a system have been identified, and fully populated. At the very least, identifying the complete set of Client-Supplier interactions manually would require significant work. Conversely, it may be possible to develop tools to log Client-Supplier interactions. However, the tool could only be run once the system was up and running (so causing significant delay to test development, and regression test identification), and

would only log interfaces actually exercised (leading to the problem of ensuring that all the relevant functions and interfaces were actually exercised).

Where the integrator is the developer, the need to develop and specify safety contracts will be understood, can be included within the project schedule and can be controlled. For non-integrator developers, contractual means can be used to encourage the development and specification of safety contracts. However, it is likely that a number of potential developers will not wish to be involved in the development of safety contracts (they may consider it to be too difficult, or be concerned about liability issues). This could restrict the use of safety contracts in the development of system tests and identification of regression tests.

- **Test systems of systems on a component pair by component pair basis.** Since the testing is based in pair interactions within the system, the whole system has to be considered on a pair wise basis. This could be very slow, and time consuming. While automation of testing would help to speed the test process, it would still take a significant amount of time to physically run the tests. It might be possible to identify end-to-end scenarios that would cover more than one pair of interactions at little extra test time cost. However, identifying these scenarios would in itself add further workload to the test process.
- **Uniformly understand the meaning of values within the system safety contracts** associated with the functionality within the original versions of the Suppliers, and any updated versions of the Suppliers. For example, the developers may work in Centigrade, while the integrators work in Fahrenheit, or the Client may work in Kelvin while the Supplier works in Centigrade.

This is typically solved by good specification. However, where the developers are in one organisation and the integrators are in another organisation, experience suggests that particular attention needs to be paid to operating with this type of balanced scorecard requirement.

- **Manage the data.** Where the Supplier has been updated, it is necessary to identify the initial safety contracts and the updated safety contracts. It is then necessary to carry out the analysis of the safety contracts to identify the updated safety contracts that are compatible with the exiting tests, using the 'same contract'/'weaker pre-contract'/'stronger post contract' rules discussed in this dissertation.

This will require some form of **data management** system to manage and make available the system wide mappings between:

- the units of functionality within the system (original system and updated system),
- the services supplied by each unit of functionality (original system and updated system),
- the Clients using each of the identified services (original system and updated system),
- the safety contracts associated with each service (original system and updated system),
- the safety contract based tests associated with each service (original system), and
- the test set from which the regression tests will be drawn (original system).

Additionally, as a Supplier is updated, it will be necessary to update this associated data.

Over a period of time, more than one update may occur within a system. As a result, the data management system needs to be robust. This is a typical **configuration management** activity, requiring the infrastructure that configuration management would normally require. Under normal circumstances, the components of a safety critical system would be under some form of configuration management. For software, it may be possible to automate the identification and management of data. For other components it is probably not possible to automate the identification of data.

The data management associated with regression testing via safety contracts will come at a price. The cost of tools or processes used to manage regression testing via this data will need to be included, in and weighed against, the benefits that regression testing via safety contracts may deliver

One potential benefit of programming languages that fully support Design by Contract (e.g. Ada, Eiffel) is the embedding of pre-conditions and post-conditions within the code body. This embedding gives

some of the data management ‘for free’ (as well as automatically providing the traditional checking of pre-conditions and post-conditions at run-time, if requested). Other languages could include the pre-conditions and post-conditions as fixed format commented text, for automated reading. Both types of code could be extended to include rely condition and guarantee conditions within fixed format commented text, with automated reading to generate part of the data necessary for the analysis step. However, extensions to the languages will be required to cover rely conditions and guarantee conditions, and even then some may be indicative rather than actual.

During **analysis**, as a Supplier is updated, it will be necessary to identify the tests that form the regression test set by:

- identifying services common to both the original unit of functionality and the updated unit of functionality (i.e. the Client/Supplier/Functionality triplets),
- establishing the updated safety contracts associated with each triplet common to both versions of the unit of functionality, and
- carrying out the comparative analysis between the original safety contract and the updated safety contract

With commercial, real world, systems, the overheads would be significant.

A simple data management solution would be to use a spreadsheet. However, a spreadsheet may not scale to larger systems. With larger systems, more sophisticated solutions may be preferable. The best solution will depend on a number of factors, and in particular on the number of services provided within the system. It may be possible to automate (or partially automate) the comparisons of the new safety contracts against the original safety contracts, particularly in the case of integer based functional safety contracts. Automation of the processes surrounding this technique is identified as an area for future work.

Additionally, this analysis demonstrates a principle already established within object-oriented coding. When developing a unit of functionality, maximum reusability can be achieved through the **weakest possible pre-contract**, and the **strongest possible post-contract**. With a weak pre-contract, the widest range of input values and initial system states can be accommodated. With a strong post-contract, the widest range of Clients can accept the returned values, and the widest range of systems can accept the guarantee condition. This thinking should inform module development.

10.2.2 Process

Section 7 develops a process to use the rules identified in Section 6. Consideration of this process highlights a number of issues:

- **Client/Supplier/system compatibility**

Where the initial tests are **not directly compatible**, it would be helpful to be able to carry out an analysis of what constitutes acceptable values within the safety contract, from the point of view of what the Client and system can provide to the Supplier within the context of the Supplier’s pre-contract, and what the Client and system can accept from the Supplier within the context of the Supplier’s post-contract.

This would allow an analysis to be carried out, without the need for dynamic testing, to evaluate whether the Client, the Supplier, and the system continued to be compatible following update, with respect to the safety contracts, and what the Client and system can supply/accept. This sort of analysis could also be carried out as an early integration activity, prior to any integration work being carried out, and any testing being carried out, to analyse Client/Supplier/system interface compatibilities.

This would require a full understanding of the interfaces between a Client, a Supplier and the system. It is necessary to know what states the Client and the system can make available to the Supplier at

the time a function is called, and the states the Supplier can make available to the Client and the system after the function has terminated.

- **Process usage will give improvement feedback**

Using the process will provide a deeper insight into how the analysis can be practically managed, and is likely to lead to process improvements. In the first instance, a small (two or three module) system, with well understood modules, functions and interfaces would be preferred. This would help to establish the value of the process, without having to manage a large number of unknowns.

10.2.3 GSN safety case patterns

Section 8.1 introduces a generic GSN pattern for a safety argument to cover the way in which regression testing has been carried out for a safety critical system or a safety related system. This pattern is high level, simply saying:

- Develop some tests against the initial system, and then test the system.
- Identify some regression tests against the updated system, and then retest the system.
- Compare the test results from the updated system against the test results from the initial system.

This generic regression test pattern is widely applicable. Virtually all development projects will need to carry out regression testing at some point in the lifecycle. However, it provides no support for what is probably the hardest part of the task – identifying the regression tests.

Section 8.3 specialises the generic GSN pattern, by tailoring the generic GSN pattern to a specific regression test identification algorithm:

- identify DSRs,
- reduce the system to Client/Server pairs to deliver the DRSs,
- identify safety contracts,
- write tests to ensure DSRs are delivered within parameters established in safety contracts,

and a specific regression test identification process:

- look for pre-existing DSR delivery mechanism, where the safety contacts are still compatible.

While developing the more specialised GSN pattern in Section 8.3, it was realised that the pattern to identify DSRs and safety contracts:

- was used twice within this pattern, as two instantiation. Since it is a large pattern in its own right, it seemed unreasonable to repeat the pattern twice.
- could potentially have other uses (e.g. in a GSN argument covering requirements specification).

As a result, this was developed as a standalone module, in Section 8.2, and referenced from Section 8.3.

The module to identify DSRs and safety contracts, as developed in Section 8.2, has three component steps:

- Identify the system DSRs.
- Reduce the system to pairs of Clients and Servers that will deliver the DSRs.
- Establish the safety contracts for the Client/Server/Functionality to deliver DSRs triplets.

However, the module developed in Section 8.2 to identify DSRs and safety contracts could be further subdivided:

- the identification of DSRs is a standalone activity that could be used in a number of different situations (e.g. as part of the requirements specification phase with a project), with the associated requirement to present this activity within a number of different safety arguments

- the mapping between Clients and Suppliers to support the delivery of DSRs is a standalone activity that could be used in a number of different situations (e.g. to support a failure mode and effects (FMEA) analysis, since it identifies what might fail, and what the function call was that caused the failure).

Conversely, it is unlikely that the identification of safety contracts would ever be used independently of the other parts of the GSN module components. Further thought should be given to the best way to split these GSN patterns and the GSN module.

However, while the exact organisation of the GSN patterns is in question, the GSN patterns themselves show that a structured, defensible argument can be developed to support the use of the rules and process developed in this dissertation. With this argument, it is possible to support an argument that a modular system has been acceptably regression tested.

When read from left to right the GSN patterns are also, fundamentally, alternative representations of the process developed in Section 7. However, the GSN representations are much more biased to concrete outputs, giving a more visual representation of the final deliverables, and a pictorial tool with the potential to be used as a tracking tool, mapping progress towards the final safety deliverables (assuming the goals and evidence are completed as a project progresses).

10.2.4 Regression test selection for timing and safety contracts

In Section 9, timing restrictions and SIL requirements have been considered, as representatives of the typical integer non-functional safety contracts that may be placed on a Supplier. In both of these cases:

- the Client can be thought of as defining, on a system by system basis, what is required of the of the Supplier, while it is the Supplier that must comply with and meet the Client's requirement, or
- the Supplier can tell the Client what it can deliver, so allowing a decision as to whether the Supplier is fit for purpose.

Where the Client defines the requirement, the requirement can be 'inverted' and included in the safety contract associated with the Supplier. However, under Design by Contract, it is the Supplier that tells the integrator what it is able to deliver (rather than the Client telling the Supplier what to do).

In the case of a SIL, it is reasonable for the Supplier to describe the standards regime and SIL that has been achieved, for the Supplier itself. Integration within the system requires that the integration of the Supplier into the system also takes place at the required SIL.

However, with timing, the picture is different. Timing will depend on a number of issues (e.g. processor speed, network latency). The permutations and combinations are endless. As a result, it is highly unlikely that a Supplier post-contract could be defined to accurately cover all the possibilities. The closest possibility to this is an indicative timing, which could not be guarantees on any system. This would have some limited value by indicating possible performance, but could not be taken as absolute. As a result, the component could not be assumed to deliver the timing and would still have to be evaluated in the target environment. This is not in the spirit of Design by Contract.

11 Conclusion

This section provides an overall evaluation of this dissertation, and the dissertation's contribution to safety critical systems engineering. Limitations and further work are also identified.

11.1 Overall Contribution to Safety Critical Systems Engineering

Modular systems are designed with change in mind. Following update, regression testing should be carried out. For safety critical systems, and safety related systems, one or more safety cases will also have to be updated. Both regression testing, and the update of safety cases are resource and time intensive activities, which may serve to reduce the benefits that could be derived from modular systems.

This dissertation identifies rules that can be used to select regression tests, a process for applying these selection rules, and safety case patterns to support the updating of safety cases when following these rules. This process has a significant advantage over the heuristic techniques that may be used to identify regression tests, since this process is scientifically grounded and rigorous, and can be used to develop a defensible argument that a system has been acceptably regression tested.

11.2 Dissertation Aim

This dissertation set out to answer the following question:

Can safety contracts and the Design by Contract rules relating to modular interchange be used to guide the identification of:

- **pre-existing tests that form the regression test set,**
- **pre-existing tests that need to be updated,**
- **pre-existing tests that need to be discarded, and**
- **areas of new functionality within the updated system, for which new tests will be required**

following update to a modular system?

11.3 Dissertation Approach

This dissertation has addressed the **link** between:

- the contract matching and class substitution rules contained within **Design by Contract**, and
- the need to **identify regression tests** for **modular systems**, following the update of one or more modules.

Design by Contract provides a framework for considering any unit of functionality as a potentially interchangeable module, where the module interfaces are specified within a contract in the form of a pre-condition and a post-condition. Safety contracts are used to extend Design by Contract into the safety domain, by placing rely conditions and guarantee conditions on functionality developed to satisfy derived safety requirements, so including system states within the contractual obligations.

This dissertation has focused on:

- **Supplier regression test selection**, via functional safety contracts based on integer clauses, where the Supplier within a Client/Supplier pair has been updated.

This dissertation also considered:

- **Non-functional timing based restrictions**, within the context of safety contracts based on integer clauses, and
- **Non-functional SIL based restrictions**, within the context of safety contracts based on integer clauses.

11.4 Dissertation Outcome

This dissertation **develops rules** that will allow the identification of:

- Regression tests from the pre-existing test set used against the original system, where there is direct compatibility between the pre-existing tests, and the module interactions/functionality contained within the updated system. This is the **directly compatible regression test set**.
- Tests from the pre-existing test set used against the original system, where there is indirect direct compatibility between the pre-existing tests and the functionality contained within the updated system (i.e. the Client/Server/Functionality triplet remains), but the interfaces have changed such that the tests need updating prior to use. This is the **indirectly compatible regression test set**, where the underlying functionality remains and can be tested, but the interfaces have changed in such a way as to invalidate the original tests.
- Tests from the pre-existing test set used against the original system, where the module interactions and/or the functionality contained within the original system no longer exist in the updated system. These tests can be discarded. These **tests are incompatible** with the new system.
- New module interactions and/or the functionality contained within the updated system that were not present within the original system, and will require new tests to be written before the newly identified module interactions and/or the functionality can be tested. These **tests must be created**, since the functionality (and therefore the need to test) did not exist in the original test set.

A **process has been identified to apply the test identification rules**.

Furthermore, argument and evidence in the form of **safety case patterns and a safety case module** have been developed for:

- **generic regression testing**,
- **regression testing based on safety contracts and the rules developed** within this dissertation, and
- **the development of derived safety requirements and safety contracts**.

This dissertation has **broad applicability**:

- across a range of **system domains** (including the non-safety domain),
- at **any level within the system-of-systems** hierarchy (e.g. system, sub-system, sub-sub-system),
- for systems comprised of a number of **different component types and architectural structures** (e.g. software, hardware, mechanical or platform), and
- for **functional and non functional requirements** on the system being changed.

It is therefore **applicable to safety critical systems and safety related systems**.

11.5 Costs and Benefits

The main costs and benefits of this technique are that:

- The **development work must be based on Design by Contract**. This is a rigorous method for the specification of interfaces. As a result of this, Design by Contract has the potential to deliver project wide benefits, by mitigating against interface incompatibility, so reducing integration errors and their associated costs. The use of Design by Contract during development also provides a solid basis for a

safety case argument about the development process. However, contracts are difficult to write, and training may be required for the integrator staff. Also, the third party suppliers of modules for integration may be reluctant to work in this way.

- The technique for regression test identification through the comparison of safety contracts is **scientifically grounded and rigorous**. This allows for a well argued safety case to be developed, which should make acceptance easier.
- The comparison of the safety contracts associated with the before update and after update Client/Supplier/Functionality triplets is **objective and sensitive** to small changes in the triplets or contracts. As a result of this, it is well suited to dealing with small as well as large incremental changes within a system.
- However, there is an **additional overhead** in identifying the Client/Supplier/Functionality triplets, and managing the comparisons of initial and updated safety contracts. This additional cost needs to be offset against the identified benefits.

11.6 Limitations

This work is limited to **regression testing**:

- Systems reduced to simple, **two module, Client/Server interactions**, where the Supplier has been updated,

and the **safety contracts** are based on

- **Functional, integer clauses,**
- Non-functional, **integer clauses relating to timings,** and
- Non-functional, **integer clauses relating to SILs.**

A **simple working example** has been used for demonstration purposes. This work has not been evaluated with a real world system.

11.7 Further Work

The following future work has been identified:

- Test the ideas presented here on a real world system

This work uses a small working example, to test the basic ideas presented here. Further work is needed to test the ideas presented here on a real world system. This would test the:

- scalability of the process, and
 - validity and practicality of the safety case pattern.
- Expand the types of regression tests being identified

This work has focused on a very limited number of safety contract clause types. Further work needs to be carried out with a wider range of values (e.g. strings, Booleans, and user defined, complex) and types (e.g. a wider range of functional and non-functional clauses).

- Investigate the effects and risks of incomplete and incorrect safety contracts

As discussed in Section 2.1.7, writing good contracts takes experience and practice. Writing incomplete and incorrect safety contracts is a credible issue. Further work needs to be carried out to

identify the risks associated with incomplete and incorrect safety contracts, and techniques for removing, reducing and mitigating these risks.

- Investigate other information that can be obtained by applying the rules.

This dissertation has focused on identifying regression test where the tests can be reused without any alterations. However, the rules developed in this dissertation also identify tests where the functionality continues to be used, possibly by different Client/Supplier pairs, or with updated safety contracts in place. Further work is required to evaluate the possibilities further, and identify any benefits to be gained from being able to identify test cases that can be reused with minimal rework.

- Investigate the use of this technique in parallel systems.

This dissertation has assumed sequential processing. However, parallel systems are increasingly common. A parallel system allows emergent properties to develop as the result of a function being executed in one part of the system, while another function is being executed in another part of the system. Further work is required to determine if and how this technique could be used on parallel systems.

- Investigate the use of tools as a means of offsetting the extra work the rules and process entail

This work may offer a way forwards by offering a rational route for the selection of regression tests. Within the context of safety critical and safety related systems, being able to offer a compelling argument that the complete and true regression test set has been identified is valuable when having to gain re-acceptance following update to a modular system.

However, the rules and process requires significant extra work for the identification and comparison of Client/Supplier/Functionality triplets. Tools may be helpful, to support these activities, and should be further investigated.

- Carry out a cost benefit analysis

As discussed above, the rules and process require significant extra work for the identification and comparison of Client/Supplier/Functionality triplets. This will add an overhead to any project using the rules and process. A cost benefit analysis should be carried out to evaluate the ideas presented in this dissertation and check that the benefits do outweigh the costs.

- Expand the work to cover situations where more than two modules interact within a function call

This work has focused on two modules interacting together (i.e. a single Client calling a single Supplier), with no outward cascades of functional calls (e.g. the initial Supplier becoming a Client, and calling another Supplier). Two modules interacting together are not the only type of functional call patterns that happens in real systems.

Being able to work with these more complex situations would be helpful, particularly if the need to identify every interaction between each and every pair of Client/Supplier modules within the system could be removed.

- Identify the best way of splitting the GSN patterns and module developed in Section 8

As discussed in Section 10.2.3, the GSN patterns could be split out in a number of ways. These ways should be investigated, and acted upon.

12 References

- 1 B Meyer, Applying "Design by Contract", Computer, pages 40-51, 1992
- 2 A Kolawa, Discovering Design by Contract, <http://www.sdtimes.com/editorial/opinion-20010915-01.html>, accessed 26May2007
- 3 ECMA International, Eiffel: Analysis, Design and Programming Language, ECMA 367, 2nd Edition, June 2006, <http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-367.pdf>, accessed 27May2007
- 4 M Dorfman & RH Thayer, Software Engineering, Wiley-IEEE Computer Society Press, Chapter 1, November 1999, ISBN 978-0-8186-7609-3, http://media.wiley.com/product_data/excerpt/94/08186760/0818676094.pdf, accessed 27May2007
- 5 Scientific America, Software's Chronic Crisis, September 1994, <http://www.cis.gsu.edu/~mmoore/CIS3300/handouts/SciAmSept1994.html>, accessed 06June2007
- 6 Eiffel Software, Building bug-free O-O software: An introduction to Design by Contract, <http://archive.eiffel.com/doc/manuals/technology/contract/page.html>, accessed 16May2007
- 7 R Mitchell & J McKim, Design by Contract by Example, Addison-Wesley, 2002, ISBN 0-201-63460-0
- 8 C Hollinshead, Cardiff Date/Time Cluster, <http://eiffelzone.com/esd/cdtc/index.html>, accessed 28May2007
- 9 Jim Weirich, Design by Contract and Unit Testing, <http://onestepback.org/index.cgi/Tech/Programming/DbcAndTesting.html>, accessed 28May2007
- 10 JC McKim, Advanced Programming By Contract: Designing for Correctness, TOOLS Pacific, 1999
- 11 CA Hoare. An axiomatic basis for computer programming. Communications of the ACM,12:576 p580-583, October 1969
- 12 G Booch, Object oriented design with applications, Benjamin/Cummings, 1991, ISBN 0-8053-0091-0
- 13 I Jacobson, Software Engineering, A Use Case Driven Approach, Addison-Wesley, 4th printing, ISBN 0-201-54435-0
- 14 B Meyer, Object-oriented Software Construction, Prentice Hall, 1988
- 15 K Waldén & J-M Nerson, Seamless Object-Oriented Software Architecture, Prentice Hall, ISBN 0-13-031303-3, http://www.bon-method.com/book_print_a4.pdf, accessed 07July2007
- 16 B Meyer, The importance of the Class Invariant, <http://web.archive.org/web/20020811203042/www.elj.com/eiffel/bm/invariants/>, accessed 07July2007
- 17 J Runbaugh, I Jacobson & G Booch, The Unified Modeling Language Reference Manual, Addison-Wesley, 1999, ISBN 0-201-30998-X
- 18 Unified Modeling Language Specification, ISO/IEC 19501:2005(E), version 1.4.2, January 2005
- 19 J Holt, UML for systems engineering, Institution of Electrical Engineers, ISBN 0 85296 105 7, 2001

- 20 Object Constraint Language, OMG Available Specification, Version 2.0 formal/06-05-01, Object Management Group, <http://www.omg.org/docs/formal/06-05-01.pdf>, accessed 14/07/2007
- 21 DF D'Souza & AC Wills, Objects, Components, and Frameworks with UML The CatalysisSM Approach, Addison-Wesley, 1999, ISBN 0-201-31012-0, <http://catalysis.org/books/ocf/>, accessed 14/07/2007
- 22 J McDermid, Trends in Safety: A European View?, 7th Australian Workshop on Safety Critical Systems and Software, Adelaide (SCS'02), Australia. Conferences in Research and Practice in Information Technology Vol. 15, P Lindsay, Ed., <http://crpit.com/confpapers/CRPITV15McDermid.pdf>, accessed 06/10/2007
- 23 Nick Tudor, Improved Flexibility in Certification, QUINETIC/KI/TIM/CR041282, 3rd May 2004, http://www.qinetiq.com/home_timpa/publications/research_outputs/improved_flexibility.Par.0001.File.pdf, accessed 08/10/2007
- 24 PM Conmy, Safety Analysis of Computer Resource Management Software, PhD thesis, University of York, 2005, <http://www.cs.york.ac.uk/ftplib/reports/YCST-2006-07.pdf>, accessed 08/10/2007
- 25 New York State Project Management Guidebook, Release 2, <http://www.oft.state.ny.us/pmmp/guidebook2/SystemAcceptance.pdf>, accessed 24/10/2007
- 26 APM Body of Knowledge, Association of Project Management, 5th edition, 2006, ISBN 1-903494-13-3
- 27 Ministry of Defence, Safety Management Requirements for Defence Systems, DEF-STAN 00-56, UK Ministry of Defence, 2004, Issue 3
- 28 Railway applications. Communication, signalling and processing systems. Safety related electronic systems for signalling, BS EN 50129: 2003, published 7 May 2007, ISBN 0 580 41814 6
- 29 NG Leveson, Safeware, Addison-Wesley, ISBN 0-201-11972-2, 1995
- 30 J Rushby, Modular Certification, CSL Technical Report, SRI International, September 2001 with revision in June 2002, <http://www.csl.sri.com/users/rushby/papers/modcert.pdf>, accessed 01/11/2007
- 31 E Schoitsch, E Althammer, H Eriksson, J Vinter, L Goeczy, A Pataricza & G Csertan, Validation & Certification of safety-Critical Embedded Systems – the DECOS Test Bench, <http://home.mit.bme.hu/~gonczy/publications/safecomp2006.pdf>, accessed 01/11/2007
- 32 P. Conmy, J. McDermid & M. Nicholson, Safety assurance contracts for integrated modular avionics, 8th Australian Workshop on Industrial Experience with Safety Critical Systems and Software, October 2003m <http://www-users.cs.york.ac.uk/~mark/papers/Conmy.pdf>, accessed 02/11/2007
- 33 Z Stephenson, M Nicholson & J McDermid, Flexibility and Manageability of IMS Projects, 24th International System Safety Conference, Albuquerque, New Mexico, USA, <http://www-users.york.ac.uk/~zrs1/pub/snm2006.pdf>, accessed 02/11/2007
- 34 B Dobbing, Practical Guide to Certification and Re-certification of AavA Software Elements, http://www.ams.mod.uk/content/docs/avsaf/pld_guid.pdf, accessed 02/11/2007
- 35 CS Pasareanu, MB Dwyer & M Huth, Assume – Guarantee Model checking of Software: A Comparative Case Study, 6th International SPIN Workshop on Practical Aspects of Model Checking, 1999, <http://pubs.doc.ic.ac.uk/model-checking-stubs/model-checking-stubs.pdf>, accessed 03/11/2007

- 36 C Blundell, D Giannakopoulou & CS Pasareanu, Assume – Guarantee Testing, Specification and Verification of Component-Based Systems, FSE 2005 workshop, September 2005, <http://www.cis.upenn.edu/~blundell/agtesting-savcbs2005.pdf>, accessed 03/11/2007
- 37 N Storey, Safety-Critical Computer Systems, Prentice Hall, ISBN 0-201-42787-7
- 38 RV Binder, Testing Object Oriented Systems: Models, Patterns and Tools, Addison Wesley, 1999, ISBN 0201809389
- 39 EW. Dijkstra, The Humble Programmer, ACM Turing Lecture 1972, <http://www.cs.utexas.edu/users/EWD/transcriptions/EWD03xx/EWD340.html>, accessed 10/11/2007
- 40 Federal Aviation Administration, Object-Oriented Technology Verification Phase 2 Handbook – Data Coupling and Control Coupling, DOT/FSS/AR-07/19, August 2007, <http://www.tc.faa.gov/its/worldpac/techrpt/ar0719.pdf>, accessed 10/11/2007
- 41 ISG of Software: Engineering Terminology, IEEE Std 610.12-190, IEEE Standard Collection
- 42 D Sundmark, A Pettersson, H Thane, Regression Testing of Multi-Tasking Real-Time Systems: A Problem Statement, SIGBED Review, vol 2, nr 2, p31-34, ACM Press, March, 2005, <http://www.mrtc.mdh.se/publications/1249.pdf>, accessed 12/11/2007
- 43 G Meyers, The Art of Software Testing, John Wiley & Sons, 1979, ISBN 0471043281
- 44 H Agrawal, JR Horgan, EW Krauser & SA London, Incremental Regression Testing, <http://www.cs.arizona.edu/classes/cs620/fall06/Papers/icm93-1.pdf>, accessed 16/11/2007
- 45 BW Weide, “Modular Regression Testing”: Connections to Component-Based Software, http://www.sei.cmu.edu/pacc/CBSE4_papers/Weide-CBSE4-7.pdf, accessed 16/11/2007
- 46 RTCA/EUROCAE, DO-297, Integrated Modular Avionics (IMA) Development Guidance and Certification Considerations and Glossary Definition, August 2005
- 47 OMG Systems Modeling Language (OMG SysML™), v1.0, September 2007, <http://www.omg.org/docs/formal/07-09-01.pdf>, accessed 19/11/2007
- 48 P Conmy, M Nicholson & J McDermid, Identifying Safety Dependencies in Modular Computer Systems, ISSC Ottawa, August 2003, <http://www-users.cs.york.ac.uk/~mark/papers/ISSC-03-dependencies.pdf>, accessed 21/11/2007
- 49 I Sommerville, Software Engineering, Addison Wesley, 4th Edition, 1992, ISBN 020165293
- 50 MA Ould & C Unwin, Testing in Software Development, British Computer Society Monographs in Informatics, 1986, ISBN 0 521 33786 0
- 51 TL Graves, MJ Harrold, L-M Kim, A Porter & G Rothermel, An Empirical Study of Regression Test Selection Techniques, ACM Transaction on Software Engineering and Methodology, Vol 10, No 2, April 2001, <http://www.cc.gatech.edu/aristotle/Journalpdffiles/tosem00-rtest.pdf>, accessed 16/06/2008
- 52 TP Kelly & RA Weaver, The Goal Structured Notation – A Safety Argument Notation, <http://www-users.cs.york.ac.uk/~tpk/dsn2004.pdf>, accessed 16/06/2008
- 53 TP Kelly & JA McDermid, Safety Case Construction and Reuse Using Patterns, <http://www-users.cs.york.ac.uk/~tpk/patterns.pdf>, accessed 16/06/2008

- 54 University of York Department of Computer Science, Foundation in System Safety Engineering, 6 – 10 October 2003
- 55 IEC, IEC 61508, Functional safety of electrical/electronic/programmable safety-related systems, May 2005
- 56 TP Kelly, Managing Complex Safety Cases, <http://www-users.cs.york.ac.uk/~tpk/sss03.pdf>, accessed 30/08/2008
- 57 TP Kelly, Using Software Architecture Techniques to Support the Modular Certification of Safety-Critical Systems, <http://www-users.cs.york.ac.uk/~tpk/scs2006.pdf>, accessed 30/08/2008
- 58 J Fenn, R Hawkins & TP Kelly, Safety Case Composition Using Contract Refinements based on Feedback from an Industrial Case Study, <http://www-users.cs.york.ac.uk/~tpk/ssscontracts.pdf>, accessed 06/09/2008