A Contract-based Approach to Designing Safe Systems

Iain Bate, Richard Hawkins and John McDermid

The Department of Computer Science, The University of York,
Heslington, York, YO10 5DD, United Kingdom

{iain.bate, richard.hawkins, john.mcdermid}@cs.york.ac.uk

Abstract. Architectural based approaches to designing software are motivating changes in the way software is developed for safety-critical systems. These new approaches allow developers to divide system requirements between components and their sub-components. To allow systems to be designed in this way a component's obligations to other components and interfaces with the rest of the system must be captured. This can be done using contracts. Other work has shown how contracts can be used to develop systems. In this paper we explore how to generate *design* and *safety* contracts, and then how to use them to support change and reuse.

1 Introduction

Safety-critical systems are those that have a direct influence on the safety of their users and the public. For such systems it is imperative that design and safety processes are conducted concurrently so that each activity can account for each other without false assumptions being made. Perhaps more important is the integration of design and safety processes.

Extensibility, adaptability, and resilience to change are increasingly desirable characteristics for many safety critical systems. Modular system architectures, such as Integrated Modular Avionics (IMA) in the aerospace sector, are being seen by many as offering the potential benefits of improved flexibility in function allocation, reduced development costs and a means of managing the ever present issues of technology obsolescence and update.

The characteristics desirable in these systems (such as change resilience and timeliness) cannot be easily retrofitted into a system design. Ability to exhibit these characteristics depends to a large extent on the architecture of the system in question (e.g. concerning the partitioning, communication mechanisms and scheduling policies). Therefore consideration must be given *during* architecture definition as to how these objectives will be satisfied. In this paper we highlight how dependability and maintainability criteria can be elaborated and considered during the architecture definition process. In particular, we describe how the exploration of alternative satisfaction arguments for these criteria can enable assessment of architectural tradeoffs.

One of the most significant problems posed by the adoption of modular systems in safety critical applications lies in their certification. The traditional approach to certification relies heavily upon a system being statically defined as a complete entity and the corresponding (bespoke) system safety case being constructed. However, a principal motivation behind IMA is that there is through-life (and potentially run-time) flexibility in the system configuration. An IMA system can support many possible mappings of the functionality required to the underlying computing platform. To have a safety case for each configuration would be infeasibly expensive and would reduce the benefits of the modular design.

Previous work has addressed the need for modular safety arguments and how this links to design trade-off analysis (Bate and Kelly 2003). However for the approach to be successful, it is also necessary to control the interfaces between different parts of the system's design and safety analysis in an integrated manner. Moreover, it is also essential that the safety requirements derived from top-level hazards are allocated to individual parts of the design to prevent the system safety analysis being *whole* system and hence the benefit of modularity being lost. A promising method of achieving this is to have contracts between the parts. For this to be successful these contracts have to be established and broken down into individual requirements placed on the parts of the system.

The contribution of this paper is to combine a trade-off analysis approach to reach an optimal design solution with a safety analysis approach to ensure the safety of a system. From these analyses, *design contracts* and *safety contracts* are established. The contracts capture dependencies between elements of the system design, i.e. what service the element provides and what each element relies upon to provide the intended service. These contracts enable documentation to be produced allowing for the effective change and maintenance of software components. The safety contracts can also be used as the basis for including safety considerations in the trade-off analysis.

Section 2 of this paper presents the approach to trade-off analysis that has been developed. The objective of managing change is one of the objectives that can be used as part of the trade-off analysis, and is of particular relevance to the work presented in this paper. Therefore, section 3 presents a decomposition of this objective using the described approach, and establishes some design options that may support managing change. A small

example system is presented in section 4 and the design options are assessed, using assessment criteria derived through the argument in section 3, from which some design contracts are proposed. The derivation of contracts through safety analysis to ensure the safety of the system and help support change is presented in section 5. The utilisation of safety contracts for safe handling of change and reuse is demonstrated in section 6. Finally section 7 presents the conclusions.

2 Architectural Design Processes

In (Bate and Audsley 2002) and (Bate and Kelly 2003) our method for architectural trade-off analysis for use within a systems engineering process was presented. Figure 1 provides a diagrammatic overview of the proposed method, which is explained further in section 2.1. It should be noted that the proposed approach could be used within the nine-step process of the Architecture Trade-Off Analysis Method (ATAM) (Kazman, Klein et al. 2001). The trade-off analysis technique has been extended from that which has previously been presented to show how design contracts are extracted and how our method for deriving safety contracts can be integrated.

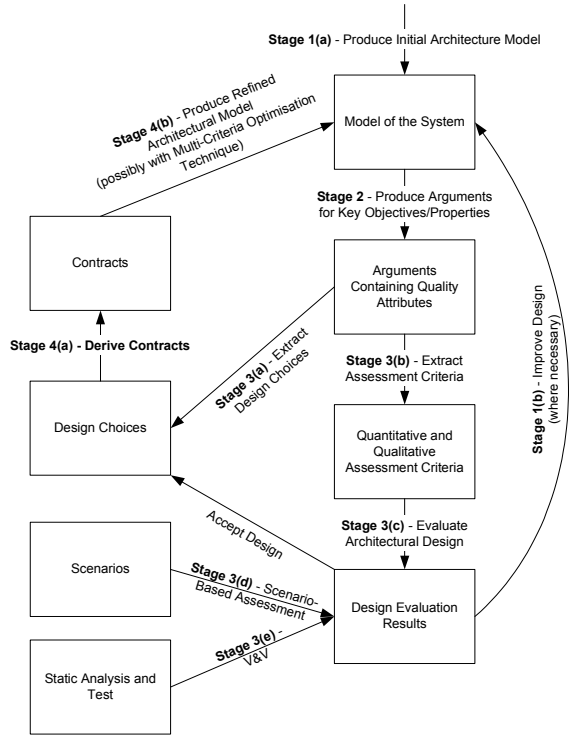

Figure 1: Overview of the Method

A key difference between our strategy and other existing approaches, e.g. ATAM, is the way in which quality attributes are derived. (Quality attributes are assessment criteria used to evaluate solutions, e.g. does the design support predictability?). The approach taken to deriving the attributes reuses techniques from the safety domain that offer strong traceability, the ability to capture design rationale and allows safety arguments to be reused.

2.1 Overview of the Technique

Stage (1) of the trade-off analysis method is producing a model of the system to be assessed. This model should be decomposed to a uniform level of abstraction. Currently our work uses class diagrams from UML for this purpose, however it could be applied to any modelling approach that clearly identifies components and the interfaces between the components.

In stage (2), arguments are produced that decompose the key objectives and properties of the system into more detailed design claims to be satisfied, along with the appropriate context for the claims, and identifies where design choices are available. The arguments are produced using Goal Structuring Notation (Kelly 1998) -refer to section 2.2 for further details. The properties of interest include; lifecycle cost, dependability, and maintainability. Clearly these properties can be broken down further, e.g. lifecycle cost into development, future upgrades and maintenance. Objectives of interest include; managed change, ease of integration and ease of verification.

Stage (3) then uses the information in the argument to derive design and verification options, and to determine assessment criteria to judge whether a particular design solution means the system meets its objectives. Other approaches for deriving assessment criteria from systems objectives include Goal Question Metrics (GQM) (Basili and Rombach 1988), and Quality Function Deployment (QFD) (Kogure and Akao 1983). Initially when the design is in its early stage the evaluation may have to be qualitative in nature but as the design is refined then quantitative assessment may be used where appropriate. Part of this activity uses representative scenarios to evaluate the solutions.

Before stage (4) of the process, based on the findings of stage (3) the design is modified to fix any problems that are identified – this may require stages (1)-(3) to be repeated to show how the revised design is appropriate. When deciding on design solutions, the results from more than one assessment criteria have to be traded-off because a design modification that suits one assessment criterion may not suit another. For example, introducing an extra processor may reduce the load across the processors in the system making task schedulability easier. However it may increase the load on the communications bus making message schedulability more difficult and increasing power consumption.

When the design modification process is complete and all necessary design choices have been made, stage 4(a) of the process extracts design contracts from the arguments and safety contracts using the safety analysis technique presented in section 5. Then, as part of stage 4(b) of the process, the process returns to stage (1) where the system is decomposed to the next level of abstraction using guidance from the arguments. Components reused from another context could be incorporated as part of the decomposition. Only proceeding when design choices are complete (and any identified problems are fixed) is preferred to allowing trade-offs across components at different stages of decomposition because the abstractions and assumptions are consistent.

Figure 2 illustrates how a component from a higher-level can be broken down in a number of ways and how the trade-off analysis should be performed across the options to determine which is the *"best"* solution. To minimise the effort required when applying this method to large systems, contracts are only defined across the interfaces of the *"best"* solution as shown in the figure.

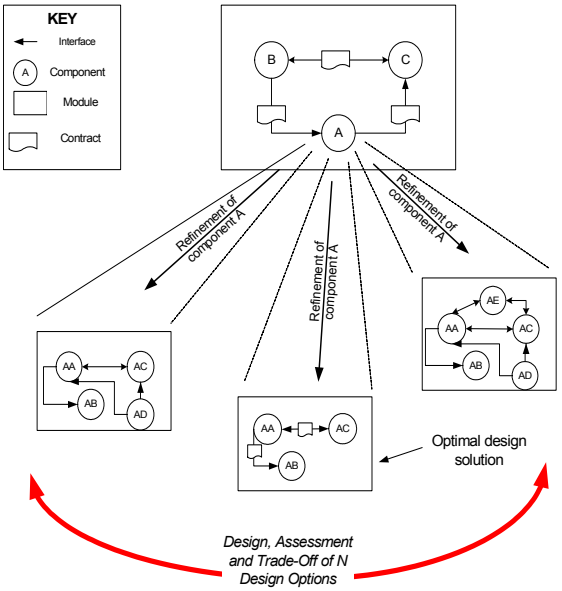

Figure 2: How the Components are Decomposed

2.2 Background on Goal Structuring Notation

GSN (Kelly 1998) is widely used in the safety-critical domain for making safety arguments. Any safety case can be considered as consisting of requirements, argument, evidence and definition of bounding context. GSN - a graphical notation - explicitly represents these elements and (perhaps more significantly) the relationships that exist between these elements (i.e. how individual requirements are supported by specific arguments, how argument claims are supported by evidence and the assumed context that is defined for the argument).

The principal symbols in the notation are shown in Figure 3 (with example instances of each concept).

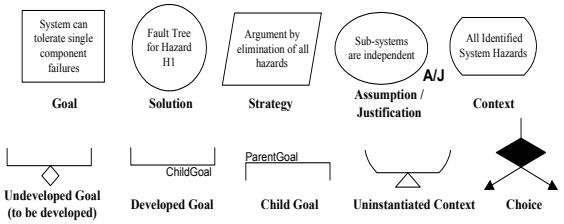

Figure 3: Principal Elements of the Goal Structuring Notation

The principal purpose of a goal structure is to show how **goals** (claims about the system) are successively broken down into sub-goals until a point is reached where claims can be supported by direct reference to available evidence (**solutions**). As part of this decomposition, using the GSN it is also possible to make clear the argument strategies adopted (e.g. adopting a quantitative or qualitative approach), the rationale for the approach (**assumptions**, **justifications**) and the **context** in which goals are stated (e.g. the system scope or the assumed operational role). For further details on GSN see [3].

3 Argument for Reducing the Work Needed to Support Changes

The role of this section is to present a decomposition of the management of change objective using the described approach, this establishes some design options that may support this objective. The design options are later supplemented to support safety using the analysis approach in section 5. In (Bate and Kelly 2003) a series of arguments were presented for the main top-level objectives of a system. The main objectives were considered to be reducing cost, achieving dependability goals and supporting managed change. The paper showed how the argument for managed change could be split into the following parts; increasing the resilience to change, reducing the work needed when change has to be performed, and reducing the scope of change. In this section, we use the argument for reducing the work needed when changes have occurred as an example.

Figure 12 contains the argument that expands the previously stated goal, **G14**, that the work needed to perform change is reduced. This goal is satisfied by a strategy that splits it into non-functional properties, **G41**, and functional properties, **G42**.

The non-functional properties are broken down here into three parts; timing, memory usage and reliability. Timing is then developed through two parts; assigning budgets so that all timing requirements are met and then showing the platform used allows the budgets to be met. Taking this approach allows each individual task to be reasoned about independently of one another. The result is the establishment of contracts between the platform and the tasks. This approach to scheduling and timing analysis technique is referred to as Reservation Based Timing Analysis (Audsley and Grigg 2001). Memory usage and reliability are left undeveloped but, like timing, the use of budgets could help ease the problems of change.

The functional aspects of the system can be handled through a choice of one of the following design strategies.

- having generic functions that are initialised at run-time using separate initialisation details, or
- specific functions and generic interfaces, or
- separating out the functionality that is likely to change.

A key reason for using GSN for decomposing the objectives is the way it supports the capturing of context. For example in Figure 12, we have an assumption, **A11**, that we have an understanding of what types of changes are likely and a justification, **J11**, reducing the work needed to perform changes is essential if cost effective upgrade is to be supported.

Table 1 presents a summary of the choices extracted from the argument in Figure 12 and a description of the pros and cons of each choice. The actual best choice in a particular situation will depend on the nature of the component's functionality and the types of change that need to be carried out. For instance there are only certain classes of function that can be made generic or robust to change without considerable cost and effort. However generic interfaces and separating out functionality are more universally applicable.

It should be noted that:

- a combination of the options can be employed where needed
- some options will be affected by other objectives, e.g. supporting managed change at the expense of a more complicated design might affect the ability to certify the final product.
- currently the options are chosen based on the results of assessing the design options against the assessment criteria extracted from the arguments. Other work has shown how multi-criteria optimisation may be used to explore the design space as part of evaluating which is the best solution (Audsley and Bate 2003).

Goal	Choice	Pros	Cons
G42	**G51** – Initialisation and generic function	Only the initialisation file needs altering not the code	1. Component can be represented by a generic function 2. Anticipated changes can be handled by the initialisation language. This means not only the types of changes but also their nature needs to be known. 3. The design will become more complicated leading to other difficulties (e.g. certification) which may negate any benefit
	G61 – Use a standard interface	Only the specific function needs altering	1. Interface semantics can be represented by standard interface 2. Anticipated changes can be handled by specific function. This means not only the types of changes but also their nature needs to be known.
	G71 – Separate out functionality	Only a single module needs to be updated	1. The types of changes must be known 2. Changes can be contained behind a standard interface

Table 1: Consideration of the Choice from Figure 12

4 Design Contracts

A proposed design for an aircraft Stores Management System (SMS) is shown in Figure 4. It is important to note that for the purpose of this paper this is a highly simplified and fictitious design with much of the functionality of an SMS removed. The main functional requirements of the SMS are to update the stores inventory. The inventory contains details of what stores are available, where these stores are located and their current status. The SMS will also send and receive data from the rest of the system and choose the next weapon to be deployed. It will also instigate remote operations e.g. arming of a weapon, and update health level information that can be used for maintenance and fault tolerance. It should be noted that WOW is 'Weight on Wheels' which is used to indicate when the aircraft is on the ground.

In the context of this paper, we are aiming to show how the approach described in section 2 can be used to assess whether a design meets its objectives. For the purposes of this paper, we are mainly interested in establishing contracts between components to help in the management of change.

From the arguments presented in section 3, Table 2 presents a number of assessment that can be applied to potential solutions. For a complete list of assessment criteria, refer to (Bate and Kelly 2003). This table indicates that the importance of all assessment criteria related to managed change is *Value Added* rather than *Essential*. (Essential criteria are those that have to be met, e.g. 'Timing requirements are met', whereas Value Added are those where it is advantageous if they are met, e.g. 'Timing requirements are met even if the software's execution times increase by 20%'.) In this case, the reason is the "Managed Change" objective itself is Value Added. However it does affect cost and when/if changes can be carried out during the system's operational life.

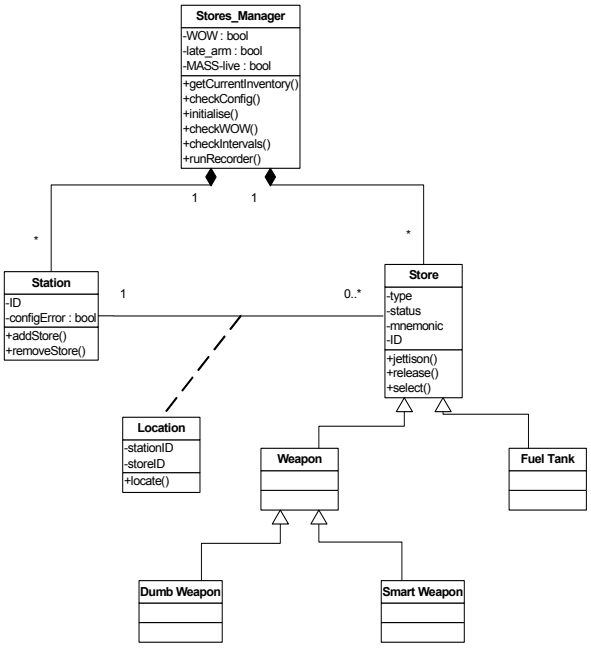

Figure 4: Class Diagram for part of the Stores Management System

Using the proposed design presented in Figure 4, Table 2 indicates the need for appropriate contracts to manage the interactions between the class *Stores_manager* and the rest of the system. Even with only a basic current knowledge of the design, the approach has allowed the general location and purpose of the contracts to be identified as shown below. The contracts will become more refined through the lifecycle, in part by using the safety analysis technique in the next section.

1. The rest of the system to provide *Stores_manager* with sufficient information so they can maintain an up to date inventory. This would require *Store* to provide type information and *station* to provide information concerning the store that it holds.
2. *Stores_manager* to correctly manage the actions of the rest of the system (e.g. which bomb to release next to maintain centre of gravity) where there is appropriate control.
3. *Stores_manager* to sufficiently mitigate the risk of hazards related to the stores. The responsibility of *Stores_manager* for avoiding hazards is on top of the self-contained mechanisms that individual stores may have.

Arg Ids	Question	Impor-tance	Response	Design Advice
G51	Can the nature of the function be made abstract of the likely/costly changes and a standard language be defined for instantiating the function?	Value Added	Information about the stores is contained within the stores and used by *Stores_Manager*. This dependency should be formalised in a contract.	This information could be stored in a central blueprint file which would be accessed using a stores id.
G61	Can an interface be produced that isolates the likely/costly changes on either side of an interaction?	Value Added	There is currently no definition of the interfaces between classes.	Use standard interfaces between classes for passing data. This interface could be kept simple especially if the majority of store specific information was held in a blueprint file.
A43	Is an accurate inventory list maintained?	Essential	Not enough design details at this level of refinement.	Establish an appropriate contract.
A44	Are the correct actions performed in relation to the stores?	Essential	Not enough design details at this level of refinement.	Establish an appropriate contract.
A45	Are key hazards related to stores avoided?	Essential	Not enough design details at this level of refinement.	Establish an appropriate contract.
G46	Does the platform allow the budgets to be met	Essential	Not enough design details at this level of refinement..	Establish appropriate contracts.
G46	Does the platform allow the budgets to be changed	Value Added	Not enough design details at this level of refinement.	Choose a platform that allows some changes in the budgets to be met.
G47	Do the budgets allow the timing requirements to be met	Essential	Not enough design details at this level of refinement.	Establish appropriate contracts.
G47	Do the budgets allow the timing requirements to be met	Value Added	Not enough design details at this level of refinement.	Choose budgets that allow some changes in the timing requirements to be met.

Table 2: Evaluation Based on Change Argument

The principal modification recommended in Table 2 is to separate out information concerning the stores in the system into a blueprint file (blueprints are basically a look-up table used in avionic systems to hold configuration information) which would allow changes to be localised and standard interfaces to be employed. This would mean that contracts would have to be established for the dependencies between the classes and the blueprint. Also as stated in item (1) of Table 2, if the current arrangement is maintained then contracts would have to be established between individual stores and *Stores_Manager*.

Table 2 also indicates that the timing properties of the system can be managed by the use of budgets as described in section 3. These budgets result in contracts being created between the tasks of the software and the platform used to execute the software. In section 6, the establishment of these contracts, based on timing requirements derived during safety analysis, is presented.

5 Safety Contracts

In this section the design solution proposed in Figure 4 is used to illustrate how safety contracts may be generated for a safety related system designed using UML. This requires that existing proven safety analysis techniques be adapted such that they can be applicable to an object oriented system. We then go on to see how these safety properties and requirements can be represented in a useful and meaningful way using the object constraint language (OCL) and how safety contracts can be used to facilitate the safe management of change of the system.

Contracts have been used in software development for many years. The principles where developed by Bertrand Meyer as the concept of 'design-by-contract' (Meyer 1988) where correctness requirements are expressed as a contract between a method and its callers. Safety contracts constrain the interactions that occur between objects, and hence can ensure system behaviour is safe. Contracts are made up of pre and post conditions. Preconditions must be true before the operation call is made and postconditions must be ensured by the execution of the call. The properties of an interaction that we are interested in from a safety perspective are function, timing and value. Analysis of each of these aspects results in requirements that are included in the safety contract. Safety is a system property and therefore, the analysis process will begin with the consideration of a system level hazard.

For the aircraft SMS in Figure 4, the initial hazard identification process identified a number of system hazards including :

- Inadvertent release of store
- Release of store whilst on the ground
- Inadequate temporal separation of store releases
- Unbalanced stores configuration
- Release of incorrect store

5.1 Functional Aspects

For all hazards identified it is necessary to perform analysis to identify how the hazard may be brought about. For this example we will look solely at the release of a

store whilst on the ground hazard. A UML sequence diagram is developed to illustrate the dynamic behaviour of the system for the relevant normal operation scenario. This can be seen in Figure 5. A fault tree is constructed using system information collected from the UML diagrams in Figures 4 and 5, and domain knowledge. This fault tree shows the failures that can occur to bring about the top event "Release of store whilst on ground". A simplified version of this fault tree can be seen in Figure 6.

It is possible to relate leaf nodes (undeveloped failure events) in the fault tree to classes in the system. For example the 'WOW not checked by store' event can be associated with the Store class in the system design. The information from the fault tree can be used to generate a definition of the hazardous behaviour of the system. This information is recorded in a table as shown in table 3.

Hazardous event (from FT)	Class	Interaction	Role	Hazardous class behaviour
WOW not checked by store	Store	checkWOW()	Client	Stores_Manager.checkWOW() call not made as required
Store releases anyway	Store	release()	supplier	Store moves to released state inappropriately
SM fails to respond to WOW signal	Stores Manager	WOW(true) – signal	supplier	Stores_Manager fails to move to WOW state

Table 3: Table of Hazardous Class Behaviour

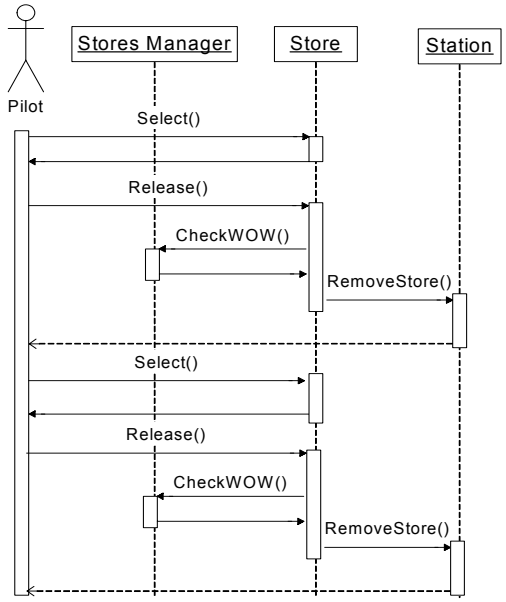

Figure 5: Sequence Diagram of Dynamic Behaviour

We have now derived hazardous conditions for classes in the system. In order to derive safety requirements and construct safety contracts for these classes it is necessary to understand how the class may behave such that these conditions can occur. To do this the state charts of these classes are studied. For the purposes of this example we will consider just the Store class. A simple state chart has been developed for this and is shown in Figure 7.

This proposed state chart design is now checked to ensure that it does not result in the hazardous behaviour identified from the fault tree, this hazardous behaviour is:

- State = release ^ ¬checkWOW
- State = release ^ WOW = true

Firstly it is assumed that the system behaves as specified in the design, that is that the class exhibits no faulty behaviour. In this simple example it can be seen from examining the state chart that this design does not exhibit any of the hazardous behaviours defined above. Checking that a proposed design does not exhibit hazardous behaviour can also be achieved using a reachability analysis tool.

This analysis has so far assumed that the class exhibits no faulty behaviour. The effects on the safety of the system if an object were to behave in an unexpected manner, that is to behave in a way other than that specified in the design, must also be investigated.

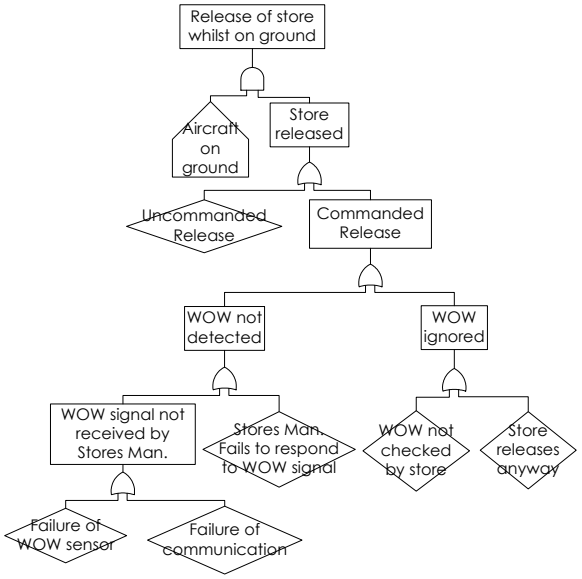

Figure 6: Fault Tree for Release of Store on Ground

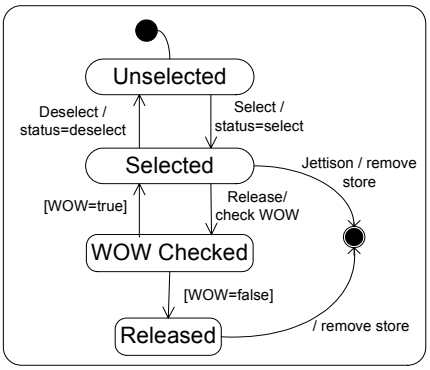

Figure 7: State Chart for Store Class

To do this we mutate the transitions in the state chart using ideas originally developed by Gorski and Nowiscki (Gorski and Nowicki 1995). Transitions in a state chart are of the general form *event[condition]/action*. The event triggers the state transition, the condition is a Boolean expression that must evaluate to true for the

transition to occur and the action is triggered when the transition fires. Transitions may have any, all, or none of these elements. In order to identify possible faulty behaviours for the transitions we can apply guidewords to each of the elements of the relevant transitions. In order to be able to simulate these faulty behaviours, extra transitions must be added to represent these deviations in the state chart. Applying the guidewords 'omission', 'commission' and 'value' to each of the elements results in five distinct transitions:

1. e[c] self-transition – event or condition is ignored
2. not e[c] / a – event spuriously generated or action performed without initiating event
3. e[not c] / a – condition taken as true when false
4. e[c] – action is ignored
5. e[c] / b (where b is an action other than a of the initiator object) – wrong action performed.

For each of the transitions in the state chart relevant to the hazardous behaviour, these five extra transitions are added to the diagram to simulate faulty behaviour. The results of this can be seen in Figure 8. It is now possible to identify if any of the faulty behaviours are unsafe. These are the faulty behaviours that can lead to the hazardous object behaviour which was defined previously.

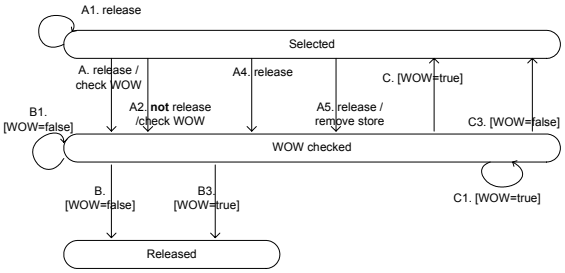

Figure 8: Mutated State Chart Showing Faulty Transitions

The faulty transitions that could lead to the hazard 'Release of store whilst on the ground' can now be analysed. The results of this are shown below.

A1. release – *Not Hazardous*

A2. **not** release / check WOW - *Not Hazardous*

A4. release – ***Hazardous*** – WOW is not checked but class may enter release state

A5. release / remove store – *Not Hazardous*

B1. [WOW=false] – *Not Hazardous*

B3. [WOW=true] – ***Hazardous*** – class enters release state when WOW is true

C1. [WOW=true] - *Not Hazardous*

C3. [WOW=false] - *Not Hazardous*

It should be noted that although many of these transitions are not hazardous, they are still faulty and would result in incorrect operation. Only the hazardous behaviour is of interest however, and so it is only this which is constrained. The system is permitted to perform incorrectly if this behaviour is non-hazardous.

There is now sufficient information about the intended and faulty behaviour of the class to begin to construct a contract for operations which may contribute to the release of a store whilst on the ground. In UML, contracts may be specified using the Object Constraint Language (OCL) (OMG 2001). OCL can be used to specify contracts using pre- and postconditions. Preconditions and postconditions are constraints defined on operations. Preconditions must be true at the moment the operation is to be executed. Postconditions must be true at the moment the operation has just ended its execution. Unlike with invariants, pre- and postconditions need only be true at a certain point in time and not all the time. These constraints will form the basis of the safety contracts. An OCL expression for an operation can be expressed as follows:

```
context
Typename::opName(param1:Type1,…):ReturnType
       pre: param1 > …
       post: result = …
```

The constraints expressed in this manner are all requirements on static aspects of the system. As can be seen with the worked example, it is often necessary from a safety perspective to express that events have happened or will happen, that signals have or will be sent, or that operations are or will be called. An extension to OCL then known as an *action clause* was proposed by Kleppe and Warmer (Kleppe and Warmer 2000) to address this problem. This has formed part of the Response to the UML 2.0 OCL Request for Proposals submission where it has become known as a message expression (Warmer, Kleppe et al. 2003). To specify that communication has taken place, the *hasSent* (^) operator is used. A simple example is given below:

```
context Subject::hasChanged()
       post: observer ^ update(12,14)
```

The post condition here results in true if an update message with arguments 12 and 14 was sent to *observer* during the execution of the *hasChanged()* operation. *Update()* is either an operation that is defined in the class of observer, or it is a signal specified in the UML model. The arguments of the message expression must conform to the parameters of the operation/signal definition. Messages in OCL are particularly useful for describing the functional aspects of the safety requirements. From the results of the analysis carried out in the example, a safety contract can be defined for the *store* class which restricts the hazardous behaviour. This safety contract is shown below:

```
context Store ::release()
       pre: none
       post: WOW=false
              and
              Stores_Manager ^ checkWOW()
```

5.2 Timing and Data Aspects

Although a large part of the safety requirements generated for any given system will be functional in nature, it is important to also consider the impact of non-functional properties on the safety of the system. Firstly the timing of the interactions is investigated. This

analysis process hinges on identifying deadlines, separations and priorities for tasks performed by the system. A task is an encapsulated sequence of operations that executes independently of other tasks (Douglass 1999). Therefore a task will consist of a number of interactions between classes in the system. Again the analysis begins with the identified system level hazards and at this point we focus on the normal scenario for releasing a store as shown in the sequence diagram in Figure 5. This scenario can be broken into the following tasks:

- Select store – This task begins with the pilot choosing a store and ends with that store being selected
- Release store – This task begins with the pilot requesting a release and ends with the store being removed from its station. This task also includes a subtask of checking WOW.

The effects of deviations on the system is investigated to identify which of these deviations may contribute to a system hazard. Firstly, tasks occurring too quickly or too slowly are considered. It should be noted that 'quickly' and 'slowly' are with respect to some undefined 'most desirable' time. It is not felt that a more concrete definition than this is necessary as this kind of assumption about interaction time is included in the model design anyway. The effect of tasks occurring too early or too late must also be considered. For this step of the analysis process it is assumed that the order in which the tasks occur for the scenario is fixed. Instead we investigate if there is a hazardous effect if the task occurs too soon after (early) or too long after (late) the previous task. Table 4 shows the result of applying these deviations to the tasks. As more details of the underlying implementation become available, more hazardous effects may become evident. This table considers deviations at the current level of abstraction.

Task	Deviation	Effect
Select Store	Quick	**No safety consequence (positive effect)** – It is desirable that the selection of the correct store occur as quickly as possible
	Slow	**Potential safety impact** – Delays in selecting the appropriate store for jettison may delay release
	Early	This task is triggered by the pilot who's decision to select a store will impact safety only if incorrect store is chosen
	Late	
Release Store	Quick	**No safety consequence (positive effect)** – It is desirable that the store be released as quickly as possible when requested
	Slow	**Potential safety impact** – A delay in releasing a store could be hazardous to the aircraft under certain circumstances
	Early	**Hazardous** – A weapon released too soon after a previous weapon could be catastrophic
	Late	**No safety consequence**

Table 4: The Effects of Timeliness of Tasks on System Hazards

Those tasks whose timeliness can have an impact on safety have now been identified. Constraints must be specified for these tasks. For quick and slow interactions it is necessary to constrain the response time of the task. If necessary a minimum response time and a maximum response time, or deadline can be specified for a task. A minimum response time will be specified for those tasks where too quick is identified as being hazardous and a deadline is specified for those where too slow could be hazardous. For tasks where early or late may be hazardous, minimum and maximum separations respectively between the completion of one task and the triggering of the next or between an event and the triggering of a task must be specified. These constraints can be used to define a safe scenario of tasks.

Domain knowledge allows us to place the following requirements on the tasks identified above as being hazardous or potentially hazardous. It should be noted that the requirements specified here are fictitious and are only used as an indication for the purposes of the example. In addition, the requirements generated makes no assumption concerning the type of scheduling and timing analysis approach adopted for the system.

- Select store – From the pilot choosing a store to that store being selected should be no longer than 200ms – ***Deadline = 200ms***
- Release store – The minimum permissible time between store releases will vary depending on the type of store being released. For this example we will specify – ***Min Separation = 100ms***
- Temporal release of store – The time from the pilot requesting a store release to that store's removal from the station should not exceed 50ms – ***Deadline = 50ms***

Up to this point only the normal scenario has been identified. A scenario is a sequence of actions that illustrates the execution of a use case. Therefore a normal scenario simply represents the normal or expected sequence of actions which occurs for a particular use case, in this example releasing a store. When considering safety however, it is important to consider alternative scenarios that may occur as these could potentially be hazardous, and may also lead to a requirement for extra timing constraints. To illustrate the scenarios clearly, the UML notation of activity diagrams can be used to show the different sequences of tasks that may realise the use case. Although activity states in an activity diagram are normally used to model a step in the execution of a procedure, here each activity state is used to represent a task or sub-task. Activity diagrams are felt to be particularly suited to this application as they emphasize the sequential and concurrent nature of the tasks in a scenario.

The alternative scenarios can be identified by omitting tasks from the normal scenario, adding in extra tasks (i.e. repetition of existing tasks), tasks occurring concurrently with other tasks or tasks occurring in an alternate order. Many of these will be of little interest and need not be considered. It is necessary to identify if any of the alternative scenarios identified could be hazardous. That is to say that they could provide an additional contribution to the hazard, they could also necessitate additional timing requirements. As part of the trade-off analysis approach, the use of scenarios to judge how well assessment criteria are met is demonstrated

As with the functional requirements, the timing requirements must be included in the safety contracts for the classes in the system. As was noted earlier, standard OCL does not provide a way of representing constraints over the dynamic behaviour of a system. An extension to OCL for modelling real-time systems has been proposed by Cengarle and Knapp (Cengarle and Knapp 2002) which provides a mechanism for representing deadlines and delays.

Deadlines for operations can be represented in the following manner:

```
context
Typename::opName(param1:Type1,…):ReturnType
    pre: …
    post:Time.now<=Time.now@pre + timeLimit
```

Where Time is a primitive data type that represents the global system time and *timeLimit* is a variable representing a time interval. In our examples we take the unit of time to be milliseconds. The above constraint represents a maximum permissible execution time equal to *timeLimit* for the operation *opName*.

Delays in reactions to signals or events can be represented in the following manner:

```
context
Typename::opName(param1:Type1,…):ReturnType
    pre:lastEvent.at+timeLimit>= Time.now
    post: …
```

Where *lastEvent.at* is the arrival time of the last event. This represents a maximum delay equal to *timeLimit* for reaction to the *lastEvent*. So based on our temporal safety analysis the following safety contract may be proposed:

```
context Store ::release()
    pre: previous_release.at+100<=Time.now
    post: Time.now <= Time.now@pre + 50
```

The data represented in the system can also contribute to system hazards if important data attributes are incorrect. It is important for each system hazard to identify which data attributes are critical. These critical data items must be constrained to ensure that they won't contribute to the hazard. It is possible to take advantage of the information hiding principle when trying to place constraints. Because the attributes of a class are private, it is only possible for them to be manipulated by operations provided by the class. It is therefore possible to protect the accuracy of data items by constraining the interactions that may manipulate that data. Again this can be done through the use of contracts.

For the system hazard 'incorrect store released' it can be identified (through a fault tree) that the pilot selecting an incorrect store, or the wrong store information being displayed to the pilot could cause incorrect store selection. This would be caused by the incorrect store being associated with a particular station. The critical attributes here are the *station* ID and *store* ID, which are associated through the location class. The only operation in our system design which can manipulate this data is the *addStore()* operation of the station class. When this operation is called on a station, the *store* ID passed as a parameter is associated with the station through the creation of a location object. By constructing a precondition for the *addStore()* operation it can be constrained to ensure the *store* ID being passed is correct. Even more so than for functional and timing aspects of systems, the data within a system is dependant on domain knowledge for deriving effective safety requirements.

6 Utilisation of Contracts

From our analysis in section 5, the safety contract for the *release()* operation of the *Store* class can be defined as:

```
context Store ::release()
   pre: previous_release.at+100 <=Time.now
   post: WOW=false
         and
         Time.now <= Time.now@pre + 50
         and
         Stores_Manager ^ checkWOW()
```

The safety contract specifies that if a service is required (in this case operation release), then to ensure this is done safely the pre-condition must be met by the client class. In return the *store* class will ensure the post-condition is achieved. If either the pre-condition or the post-condition is violated then the operation may be unsafe. When a system is designed, classes collaborate using message passing to achieve functionality. Figure 9 shows three classes collaborating in such a manner. The filled rectangles before or after the operation indicate that a pre- or post-condition respectively of a safety contract exist for that operation.

Figure 9: Interactions including pre- and post-conditions

System : SMS_Aircraft X v.1.2			
Class : Store			
Clients	***Interaction***	***Reqts. requiring satisfaction***	***Guarantees to be made***
A	Release()	`previous_release.at + 100 <= Time.now`	`WOW=false`
			`Time.now <= Time.now@pre + 50`
			`Stores_Manager ^ checkWOW ()`
Supplier	***Interaction***	***Reqts. to be met***	
W	X()	`pre-conditions of X()`	

Table 5: A Table to Capture Safety Requirements

In this situation class *A* must meet the pre-condition of *Release()* as this operation is being called by *A*'s operation *B()*. It is important to note that even though the pre-condition of *Release()* is defined in the class *store,* it becomes a derived safety requirement for class *A*. In turn, the *store* class must meet the post-condition of *Release()*, it must also however meet the pre-condition of operation *X()* defined in class *W* as the *Release()* operation makes a call to *X()*. The pre-condition of *X()* may not necessarily match the post-condition of *Release()* and therefore an extra derived safety requirement has been identified for class *Store*.

The above example has been used to illustrate that when interactions occur between classes, the requirements get 'shared out' amongst the participating classes. For the system to be safe it must be shown that all classes meet any safety requirements placed upon them.

As part of using contracts it is important to be able to show that when a system is constructed the classes 'fit together' safely. That is to say that all the requirements are picked up by the relevant classes. It is suggested that a table could be used as a way of capturing the information. Table 5 shows such a table.

This basic table specifies the system and the class in that system for which the table is being constructed. If more than one version of the system has been designed then it is important to record which version the requirements in the table relate to. As is discussed later, changes to the system design can alter the derived requirements applicable to a particular class. The table then records a list of the client classes. These are classes that make calls to services of the class. Table 5 shows the table for the *store* class from Figure 9 and therefore the only client is class *A*. The only interaction that class *A* has with the *store* class is through the *Release()* operation. The pre- and post-conditions of this operation give the 'Requirements requiring satisfaction' and 'Guarantees to be made' respectively. The table also records the supplier classes for the *store* class. These are classes whose services *store* utilises. Again there is just one supplier class in this example, class *W*. The interaction that store has with class *W* is recorded and the pre-conditions of those interactions give the 'Requirements to be met'.

Using the table constructed in table 5 the derived safety requirements of class store can be identified as being those defined in 'Guarantees to be made' plus those in 'Requirements to be met'. The derived safety requirements of the relevant client class are those defined in 'Requirements requiring satisfaction'. It is important to ensure that 'Requirements requiring satisfaction' in *store* are reflected as 'Requirements to be met' in the relevant client class (class *A*). It must also be ensured that 'Requirements to be met' in *store* are those defined in 'Requirements requiring satisfaction' in relevant supplier class (class *W*). In this way a table such as that in table 5 can be used to check that the classes in the proposed system design can work together safely. This is done by checking that all the correct derived safety requirements have been allocated between the classes in the system. Another advantage of constructing a table such as this is that it allows an individual class in a system to be developed independently of other classes. All the derived safety requirements for that particular class are explicitly laid out and further reference to other classes to elicit requirements is not necessary.

6.1 Applying Trade-Off Analysis Results to the Derived Safety Requirements

This section shows how the results of the safety analysis integrates with the trade-off analysis method to manage the objective of "Ease of Change" as well as ensure safety requirements are met

In section 2.2, the use of timing budgets to mange the problems of change was introduced. In the case of the example requirements given in Table 5, there are two timing requirements that have been derived as part of the safety analysis process. These are a separation (i.e. 100 ms) requirement and a deadline (i.e. 50 ms).

Using Reservation Based Analysis, these response requirements would be decomposed into execution timing budgets on each individual method. The difference between response time and execution time is dependent on the scheduling method employed, and hence how the method may be interfered with (by higher priority methods/tasks) or blocked (by lower priority tasks/methods). For example, the following budget may be assigned

Release() – Best-Case Execution Time >= 1 ms

Release() – Worst-Case Execution Time <= 5 ms

When choosing/changing a hardware platform, it has to be shown that the budgets for each method/task are met. Where this is not possible, either different hardware would be needed or the budgets re-allocated across tasks. In addition when "new" requirements are introduced or existing ones change, checks have to be performed to see whether the existing budgets are still valid. For the approach to be successful, the budgets assigned to each method/task have to be chosen to promote flexibility, scalability and help manage obsolescence. In (Audsley and Bate 2003) an allocation approach of this form is presented.

6.2 Handling Change

A number of changes will normally occur to the design of a system during the development process. It is preferable that the majority of these changes, particularly major architectural level changes, will happen as early as possible in the process. In this way the amount of rework required as a result of that change can be minimised. It is highly likely however that change will occur to a system after a large amount of the safety analysis effort has been performed. It is often the case that when a change occurs to a system, the amount of reanalysis effort required in order to show that the modified system is still safe is proportional to the size of the system as a whole rather than to the size of the change that has been made.

Using a modular approach enables the impact of a change to a system design to be minimised. The change tends to affect a particular aspect of a model rather than affecting the entire structure of the system, therefore the 'cost' of making a change is thus greatly reduced (Liskov and Wing 1994). If this benefit is to be realised in a safety-related or safety-critical system, it is imperative that the effect of change on the safety of the system can be 'contained' in a similar manner. This allows the required re-analysis to become more proportional to size of the change made. This can be achieved through the use of safety contracts.

If the design of a class in the system is changed then it is necessary to show that the requirements that were placed upon that class can still be met. The class may do better

or worse than it did before as long as the requirements are still met. This is the simplest form of change to deal with as the requirements placed on classes have not changed. Things are more complicated if additional interactions are introduced to the system as a result of changes. This is due to the fact that these interactions may introduce new ways in which a hazard could occur.

Figure 10 shows the system from Figure 9 but with an additional interaction introduced between the store class and class *W*. Also an extra class, *X*, has been introduced to the system. This new class interacts with both the *store* class and class *A*. It is necessary to understand the impact that these new interactions may have on the safety of the system. The impact of the interactions introduced in Figure 10 is fairly straightforward to deal with as the interactions that occur are calling operations that previously existed in the system design. Therefore no new analysis is required on these interactions as the operations would have been covered as part of the original safety analysis process. It is only required that any contractual obligations that exist on these operations are picked up as derived safety requirements on the relevant class.

For the interactions in Figure 10 that occur between class store and class W, and class *X* and class *A*, no safety contracts exists for the operations involved in these interactions, therefore no new safety requirements are derived for any of the participating classes. For the interaction between class *X* and class *store*, there exists a safety contract on the *Release()* operation. Therefore class *X*, as the client for that interaction, will acquire a derived safety requirement of the pre-conditions of *Release()*. This would be captured in a safety requirements table constructed for class *X*.

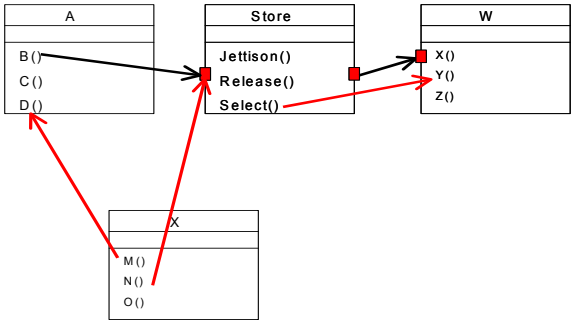

Figure 10: Introducing Additional Interactions

Figure 11 again shows the system from Figure 9 with some different changes. In this case the new interactions that have been introduced to the system are calling operations in class *X*. As this class was not part of the original system for which the safety analysis was conducted, there may be ways in which the operations of class *X* may contribute to a system hazard which have not been considered by the analysis. To ensure they don't contribute to a system hazard, these operations may require the specification of a safety contract. This will require further analysis in order to determine the nature of any safety contract. Any contract identified would lead to additional derived safety requirements. The original analysis performed on the system started from top-level system hazards and worked downwards. To minimise and localise the reanalysis required for these additional interactions, the analysis may instead be performed bottom-up. This would identify if and how that specific interaction can contribute to any of the system hazards.

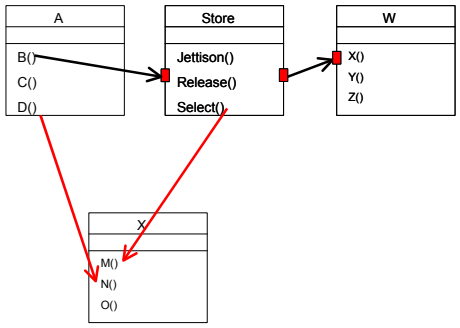

Figure 11: Introducing calls to new operations

6.3 Reuse

There is a great potential for reusing elements of a system in another similar system. This saves the expense of designing that part of the system from scratch. As with change however, it must be shown that this can be done safely. It is therefore necessary if the savings are to be realised that the relevant safety contracts can also be reused in the new system. It is not enough however just to be able to show that the existing contract is still met in the new system. As the new system will be different from the original, there may be new system level hazards which need to be considered which will not have formed part of the original analysis.

A key issue is that the context of the system may also have changed. Context will include such things operational role and physical or regulatory environment. This means that the safety contracts derived for the original system may not be appropriate in the new system. The differences in the new system will necessitate further analysis to ensure that the correct safety contracts are defined. Therefore the less the new system differs from the original system, the easier it will be to reuse it. If too much additional analysis is required due to differences in the systems, then the savings from reuse will be greatly reduced. It is important therefore that if classes or components are going to be reused successfully that as much contextual and system information as possible is captured. This information could be added to the table of requirements.

7 Conclusion

The use of contracts is essential to show how and why requirements have been decomposed. This paper has shown how contracts are generated through the use of a structured and integrated safety and design process. As part of this we have shown how different properties and objectives can be traded-off to achieve an appropriate design. A safety analysis process is then applied to the resulting design to manage the risk associated with the key hazards. This is demonstrated using a Stores Management System from an aircraft. In the example the objective of managed change is given particular attention,

and therefore both safety and design contracts and appropriate interfaces are established that allow us to meet the objective. Further to this we have shown how the contracts can be used to facilitate lifecycle upgrades and as such reduce lifecycle costs.

8 References

Audsley, N. and I. Bate (2003): Utilising Co-Design Techniques in the Hard Real-Time Systems Development and Implementation. submitted to the Co-Design Track of Real-Time Systems Symposium.

Audsley, N. and A. Grigg (2001): Reservation-Based Timing Analysis - A Practical Engineering Approach for Distributed Real-Time Systems. Proceedings of IEEE Conference on Engineering Computer-Based Systems.

Basili, V. and H. Rombach (1988): The TAME Project: Towards Improvement-Oriented Software Environments. IEEE Transactions on Software Engineering 14(6): 758-773.

Bate, I. and N. Audsley (2002): Architecture Trade-off Analysis and the Influence on Component Design. Proceedings of Workshop on Component-Based Software Engineering: Composing Systems from Components.

Bate, I. and T. Kelly (2003): Architectural Considerations in the Certification of Modular Systems. Special Issue from SAFECOMP 2002 of the Journal of Reliability Engineering and System Safety.

Cengarle, M. and A. Knapp (2002): Towards OCL/RT. Lecture Notes in Computer Science 2391: 390-409.

Douglass, B. P. (1999). Doing Hard Time - Developing Real-Time Systems with UML, Objects, Frameworks, And Patterns, Addison-Wesley.

Gorski, J. and B. Nowicki (1995): Object Oriented Approach to Safety Analysis. Proc. ENCRESS '95: 338-350.

Kazman, R., M. Klein, et al. (2001). Evaluating Software Architectures - Methods and Case Studies, Addison-Wesley.

Kelly, T. (1998): Arguing Safety - A Systematic Approach to Safety Case Management. Department of Computer Science, The University of York.

Kleppe, A. and J. Warmer (2000): Extending OCL to Include Actions. Lecture Notes in Computer Science 1939: 440-450.

Kogure, M. and Y. Akao (1983): Quality Function Deployment and CWQC in Japan. Quality Progress: 25-29.

Liskov, B. and J. Wing (1994): A Behavioural Notion of Subtyping. ACM Transactions on Programming Languages and Systems 16: 1811-1841.

Meyer, Bertrand (1988): Object-Oriented Software Construction, Prentice Hall.

OMG (2001). Object Constraint Language Specification. Unified Modeling Language Specification version 1.4, Object Management Group.

Warmer, J., A. Kleppe, et al. (2003): Response to the UML 2.0 OCL RfP - Revised Submission, Version 1.6, OMG.

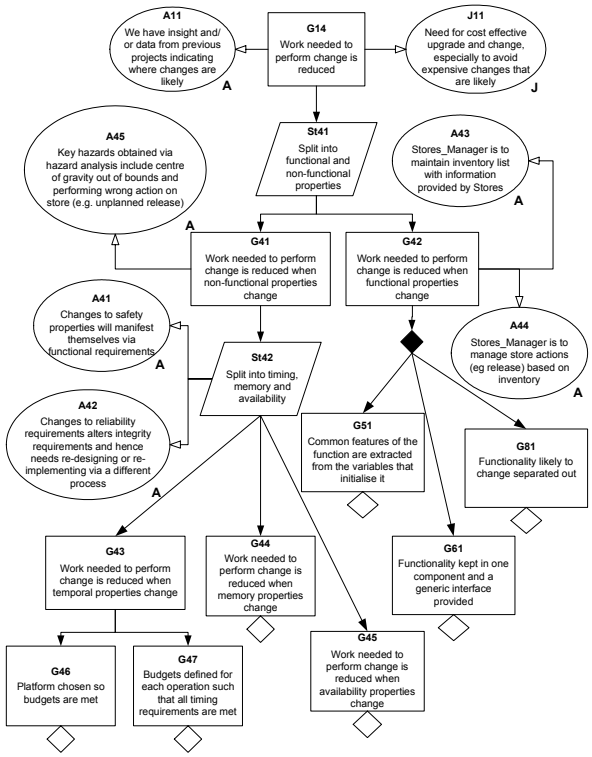

Figure 12: Argument for Ease of Change