Developing Safety Contracts for OO Systems

R.D. Hawkins; University of York; York, UK

Prof. J.A. McDermid; University of York, UK

I.J. Bate; University of York; York, UK

Keywords: OO, safety, contracts

Abstract

The use of an object oriented (OO) approach brings potentially large savings in terms of time and cost for developers of safety critical systems. OO systems have improved maintainability due to encapsulation, high cohesion and low coupling, and the facility for reuse through inheritance and design patterns. This raises specific challenges for developers of OO safety critical systems who wish to take full advantage of these features. To realise the benefits requires an ability to reason about the safety of individual classes or components in the system. This is quite difficult to achieve with most existing safety analysis techniques, as hazards tend to be decomposed down in a functional way. In this paper we initially explore how existing techniques may be adapted to provide the required results. We then go on to examine how these safety properties and requirements may be represented in a useful and meaningful way.

We propose to use safety contracts for classes as a way of capturing safety requirements in an OO system. These contracts are constructed through analysis of functional, timing and value aspects of interactions within the system. We look at how these contracts can be incorporated into the system design and then used to verify that a system is safe. We go on to explore how the use of safety contracts facilitates maintainability and reuse.

Introduction

Although the use of OO techniques has become increasingly widespread throughout the IT community, its use in safety-related applications has, up to this point, been fairly limited. However, interest in the use of OO for safety-related systems is increasing as the benefits of adopting such an approach have become apparent. One of the biggest potential benefits arises from the improved maintainability of OO systems. It is desirable that the amount of analysis required when a change is made to a system is proportional to the size of the change, rather than to the size of the whole system as is, in general, currently the case. OO provides a way of moving towards this aim by taking advantage of encapsulation, high cohesion and low coupling which characterize OO systems. Another potential benefit arises from the ability to reuse part of an existing system which does what is required, in another similar system under development. Inheritance provides a way of adding to and extending existing components, whilst requiring only the elements that are different in the new system to be developed. Design patterns, which are commonly used in OO also provide a way of capturing design solutions for common problems. This reduces the development effort required when addressing such a problem.

If any of these benefits are to be realized for safety-related systems it is necessary to be able to reason about the safety of individual classes or components in the system. Without being able to do this the benefits gained through the encapsulation and low coupling of OO designs would be lost, as the safety process would not be aligned with the design. Unfortunately with existing safety analysis techniques it is difficult to reason about individual classes or components. This is

due to the fact that the majority of techniques e.g. FFA, decompose hazards functionally. System level hazards are identified, and for each, functional failures which may contribute to that hazard are elicited. For a functionally decomposed system it is much easier to allocate a functional failure to the failure of a system component. This is because there is often a direct mapping between a functional failure and a subsystem. An OO design is not decomposed functionally, but rather into objects. The functionality in the system is realized by many objects collaborating through message passing to achieve a system function. Therefore a functional failure will not easily map onto a single element of the design. It is important therefore that existing techniques can be adapted such that failures can be allocated between objects. From the system hazards, system safety requirements are defined which are used to specify contracts between classes. These contracts can then be used to derive class safety requirements. It is also necessary that the safety properties and requirements generated as a result of the analysis can be represented in a useful and meaningful way. This includes having the ability to understand how these requirements are affected by design changes and reuse.

Safety Contracts

We propose to use *safety contracts* as a way of capturing safety requirements on classes in an OO system. Safety contracts constrain the interactions which occur between objects and hence can assure that system behavior is safe. Contracts are used to specify the relationship between the client (the operation caller) and the supplier (the called routine) as precisely as possible. This is done using assertions. Pre- and post-conditions are assertions which apply to individual routines. Preconditions express requirements that any call must satisfy if it is to be correct. Preconditions must be true before the operation call is made. The Post-condition expresses properties that are assured in return by the execution of the call. A safety contract is not intended to specify requirements for correct behavior, it is used only to specify that behavior required in order to assure the system is safe (i.e. that system hazards can not occur). These safety contracts build up a set of safety requirements for each class which could affect the safety of the system. Each interaction in which an object partakes may have a safety contract associated with it. As illustrated in figure 1, an object may be part of many contracts both as client and/or supplier. Each contract for which the object is client or supplier may generate obligations on the object. The object must satisfy either the pre- (if client) or post- (if supplier) conditions of these interactions. These conditions together form the safety requirements for that object. Figure 1 shows how the set of safety requirements for an object will be formed from pre- or post-conditions of a number of interactions.

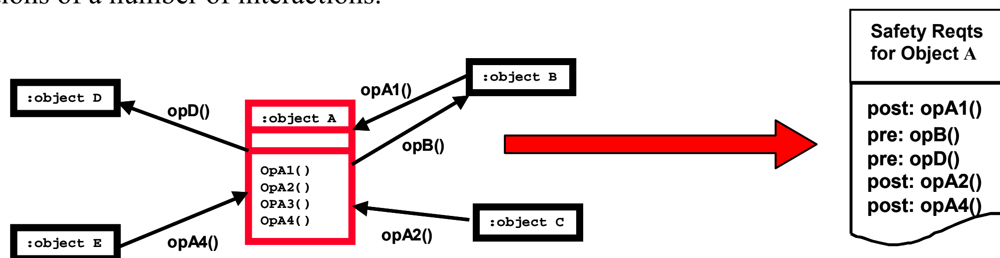

Figure 1 – Safety Contracts Define Safety Requirements for a Class

Defining Safety Contracts

In order to define safety contracts for interactions in the system, it is necessary to consider a number of different properties of those interactions to identify how they may contribute to a system hazard. Three properties of an interaction are considered, function, timing and value. For

each of these properties, safety analysis is performed to identify how these interaction properties could lead to the system hazard. The results of this analysis then define the requirements that make up the safety contract. The starting point for the analysis is a list of system level hazards identified during the initial hazard identification process. For each of the hazards identified it is necessary to perform analysis to identify how the hazard may be brought about.

<u>Function</u>: A specification of the system design is required for analyzing the behavior of the system. A UML class diagram can be used to specify the static structure of the system, the dynamic behaviour can be specified using an interaction diagram. Figure 2 shows UML class and sequence diagrams for a simplified aircraft stores management system (SMS). This example will be used throughout the paper

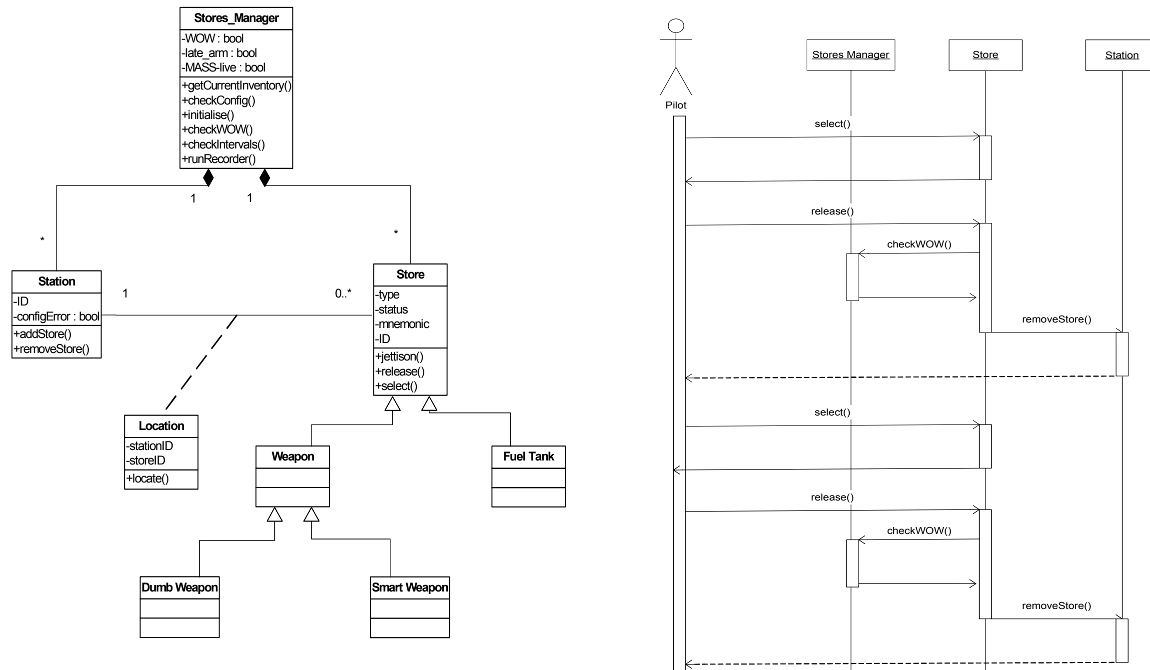

Figure 2 - UML Class and Sequence Diagram for SMS

The method for analyzing the safety of functional behavior of the system is discussed in reference 1 and is applied to the SMS example above in reference 2. This method uses fault trees to extract the following hazardous behavior for the store class in the SMS system:

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

The state chart for the store class (figure 3) is then analysed by mutating transitions between states (figure 4) to identify behaviour of a store class that could lead to the hazardous behaviour above. It is possible for more complex system designs to use an automation tool to generate the mutated transitions and to check which of these transitions may bring about the hazardous condition. Such a tool is under development at the University of York. Although most of the faulty transitions identified through this method are not hazardous (i.e. will not contribute to the system hazard) they still result in incorrect operation. As mentioned earlier, in defining a safety contract we are only interested in constraining the hazardous behaviour. 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 system hazard.

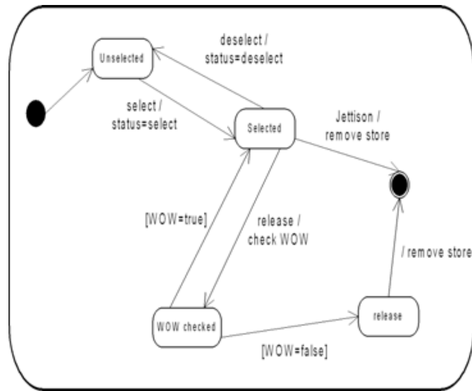

Figure 3 – State Chart for Store Class

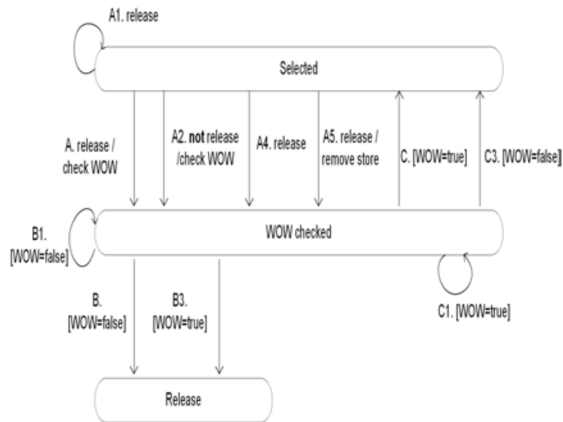

Figure 4 – Mutated State Chart Showing Faulty Transitions

Timing and Value: 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 (ref.3). 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 analysis involves investigating the effect of deviations on the tasks that are performed to identify which of these deviations may contribute to a system hazard. The deviations considered are tasks occurring too quickly or too slowly and early or late. Early and late correspond to a task occurring too soon or too long after the previous task or event. The result of applying these deviations to the identified tasks is shown in table1.

Those tasks whose timeliness may have an impact on the safety of the system have now been identified and constraints must now 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 (see figure 5).

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 1 – The Effects of Timeliness of Tasks on System Hazards

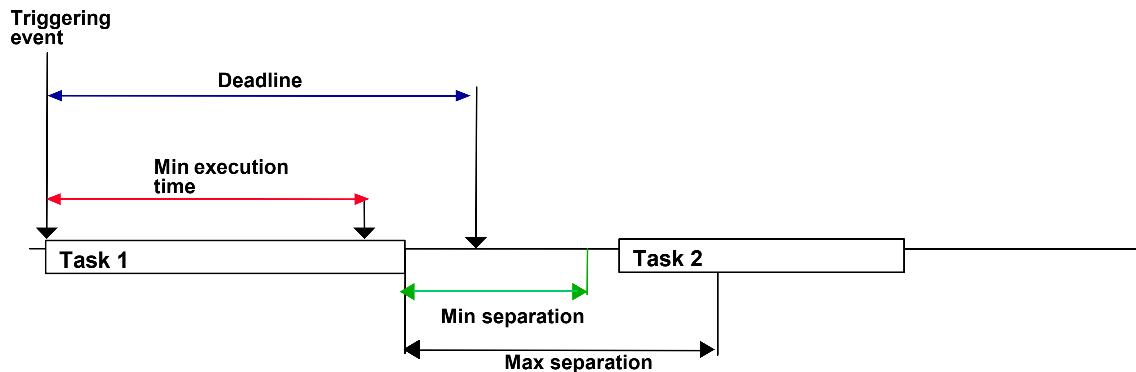

Figure 5 – Constraints applicable to a sequence of tasks

Domain knowledge is used to place the following requirements on the tasks identified above as being hazardous or potentially hazardous. It should be noted that this analysis is not trying to produce accurate estimates of execution times for operations or transactions, but to specify the minimum requirements for a safe system. Requirements should therefore be as weak as possible to allow maximum flexibility to the implemented system. In (ref. 4) we presented an approach and framework for identifying, in control systems, an appropriate and valid set of timing requirements, and their corresponding control parameters. The approach was based on decomposing the systems objectives to a number of design choices and assessment criteria. Each design choice was evaluated using a combination of static analysis and simulation. Heuristic search techniques were then used to search the design space for the design solution considered most appropriate. The figures given below are for illustration purposes only.

- Select store – From 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
- Release 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 which may occur as these could potentially be hazardous, but may also lead to a requirement for extra timing constraints. In order to illustrate the scenarios clearly, the UML notation of activity diagrams can be used to show the different sequences of tasks which 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. 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.

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 inherent in OO 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 (as identified from the class diagram in figure 2) are the station ID and store ID, which are associated through the location class. The only operation specified in this 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 assure the store ID being passed is correct. The nature of this constraint can only be properly specified with a great deal of understanding about the system under consideration. Even more so than with functional and timing aspects of the system, the data within a system is very dependant on domain knowledge for deriving effective safety requirements

Specifying Safety Contracts

Once the safety requirements for the classes in the system have been derived it is necessary to specify these in a useful and meaningful way. In UML contracts may be specified using the Object Constraint Language (OCL) (ref.5). OCL can be used to specify three types of constraints, invariants, preconditions and postconditions. Invariants state a condition that must always be met by all instances of a class, type or interface. Invariants are expressions which evaluate to true if the invariant is met, and must be true all the time. Preconditions and postconditions are defined on operations and need only be true at a specific point in time i.e. at the start or end of execution. 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. An OCL expression for an operation can be expressed as follows:

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

These constraints will form the basis of the safety contracts. One limitation with OCL is that the constraints that can be expressed are all requirements on static aspects of the system. As was seen in the example above, 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 has been proposed (ref.6) that addresses this problem with the introduction of a new type of constraint called the action clause. The action clause contains three parts: a set of target instances to which the event(s) is sent, a set of events that has been sent to this target set, and an optional condition. The action clause evaluates to true if, and only if, whenever the condition holds, the virtual output queue of the instance that executes the operation has contained at some point in time during execution of the operation, all target – event pairs that are specified in the combination of the target set and the event set. This action clause is 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 behavior. This safety contract is shown below:

```
context Store ::release()
pre: none
post: WOW=false
action: to Stores_Manager send checkWOW()
```

A further limitation of OCL is that no way is provided for representing constraints over the dynamic behavior of a system. Again an extension to OCL has been proposed for modeling real-time systems (ref.7). This provides a mechanism for representing deadlines and delays. Deadlines for operations can be represented in the following manner:

```
  context Typename::operationName(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 ms. The above constraint represents a maximum permissible execution time equal to timeLimit for the operation `operationName`.
Delays in reactions to signals or events can be represented in the following manner:

```
  context Typename::operationName(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`. The requirements derived for the timing aspects of the store class can thus also be represented as part of the contract. The safety contract for the store classs of the SMS would therefore be of the form:

```
context Store ::release()
pre: previous_release.at + 2500 >= Time.now
post: WOW=false
      Time.now <= Time.now@pre + 250
action: to Stores_Manager send checkWOW()
```

Although OCL is not the only possible mechanism for representing safety contracts, as it is part of the UML, it is more easily integrated with the system design. The safety requirements on each class in the system design are therefore explicit. For a system to be safe, the class must meet the

set of safety requirements consisting of all pre conditions of interactions for which it is the client, and all post conditions of interactions for which it is the supplier. To know that a system will be safe it is necessary to show that the complete set of safety requirements are met.

Managing Change and Reuse Through Safety Contracts

The safety requirements for the system design have now been specified. It is possible however that changes may occur to a system design at a time after these requirements have been derived. As mentioned earlier, an advantage of adopting an OO approach to system design is that there is improved stability in the presence of changes. Changes in the requirements do not tend to affect the entire structure of the system but are localized to particular aspects. This means that the "cost" of making a change can be greatly reduced. However in order to truly realize this benefit for safety-related systems the re-analysis required must also be localized as much as possible to the particular aspects that have changed. It is proposed that safety contracts provide a mechanism for achieving this. Changes can affect the system in different ways.

If a class in a system has safety requirements defined through contracts, as long as all system hazards have been correctly identified, safe behavior has been defined for the interactions in which that class partakes. If changes are then made to the class design then the class must still guarantee to do what it did before to meet its contractual requirements *or do better.* It should also not require anything more than it did before but it *may require less.* It must also still fulfill the requirements of other classes with which it interacts as a client. Once the change is made, it is sufficient therefore to show that the new class design still meets these requirements.

This is only true however if the changes that have been made to the class do not include the introduction of additional interactions with other elements of the system. If new interactions are added then this could introduce new ways in which a hazard could arise in the system. The new interaction could be a call to an existing operation with which there was previously no interaction. In this case, if a safety contract exists for this operation then the calling class must assure the newly acquired safety requirement defined by the precondition of the called operation can be met by the class design. If no safety contract has been specified for the operation called in the interaction then it is necessary to check that the interaction could not contribute to a system hazard. This will then necessitate further analysis on this new interaction to determine what new derived safety requirements are necessary. This would also apply in the case of additional classes being introduced into the system design. As well as additional interactions, the effect of interactions being removed must also be considered as this could also impact on the ability of a class to meets its contractual operations.

In an OO system, a change may also result from the use of inheritance to extend an existing class. The subclasses produced using the inheritance mechanism inherit all the attributes and operations of the parent class. Additional attributes and operations may then be added by the subclasses if required. In the same way that attributes and operations are inherited, it is proposed that the safety contracts of the parent class be inherited in a similar fashion. The new version of the contract which is created must remain compatible with the original contract, however it may improve on it by weakening the original precondition or strengthening the postcondition. These inherited contracts can be extended as required to include further derived requirements for the subclass. As with all changes made to a system it is necessary to check that the classes meet any contractual requirements placed on them. Where inheritance occurs this process can be more complicated.

Figure 6 shows an example from the SMS where the properties of store class are inherited to created two new classes, a weapon class and fuel tank. Weapon class has inherited the release

operation from class store and redeclared it (changed the implementation of that operation). The principles of polymorphism allow store objects to become attached to instances of weapon. If the stores manager were now to make a call to the release operation then dynamic binding would ensure the redeclared version of release in weapon would be called rather than store's original version. The difficulty arises because the author of the stores manager class can only look at the safety contract for the release operation in store and could thus potentially violate a safety contract for the operation that is actually being called in weapon. Fortunately the use of safety contracts means that if stores manger meets contractual obligations for release in store, then it is known that the obligations for release in weapon will also be met as the weapon class can only match or improve on the safety contract in store.

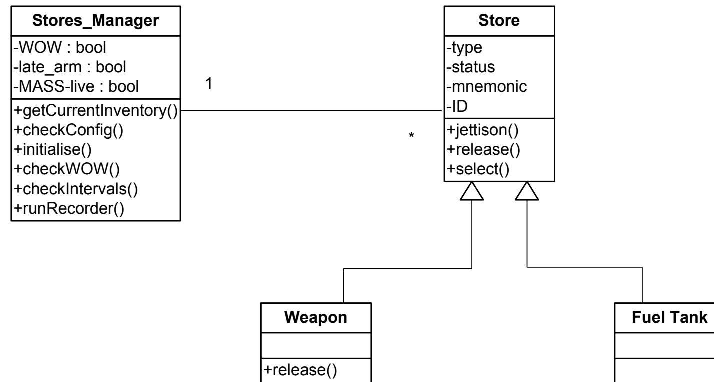

Figure 6 – Redefinition of an Operation Under Contract

Inheritance is a useful mechanism for reuse as it makes it possible to reuse an existing repertoire of classes that have behavior similar to that which is required in a new system. However when using inheritance to reuse classes in a new system, there are additional considerations. It is not sufficient to inherit existing contracts and show that they are met as described above. As the system under consideration is different from that in which the original parent class was used there may be different system level hazards to consider. Hence it is necessary to show the existing class safety requirements are still valid in the context of the new system safety requirements. The classes being reused in the new system may potentially contribute to these additional system hazards and therefore necessitate extra constraints in the contract. For example, if the store class from the example in figure 6 were used to derive weapons classes in a different aircraft, there may be additional hazards specific to that aircraft which were not present in the original system. This will require additional analysis for each of these hazards. The system in which the elements are being reused should be as similar to the original system as possible. If this is not the case then the re-analysis required to ensure the safety of the system will be large and the benefits of reusing existing elements will be greatly reduced.

Conclusions

In this paper we have described how safety contracts can be constructed for OO systems through focused safety and hazard analysis. We have then gone on to look at how OCL can be used to represent these contracts. Finally we have discussed how contracts can be used to facilitate change and reuse.

References

1. Hawkins, R. and McDermid, J., Performing Hazard and Safety Analysis of Object Oriented Systems. In Proceedings of 20th ISSC, 2002: p.802-811
2. Bate, I., Bates, S., and Hawkins, R., A Contract Based Approach to Designing Safe Systems. Submitted to SAFECOMP 2003.
3. Douglas, B.P., Real-Time UML – Developing Efficient Objects for Embedded Systems. 1998: Addison-Wesley
4. I. Bate, J. McDermid and P. Nightingale, Establishing Timing Requirements for Control Loops in Real-Time Systems, To Appear in the Journal of Microprocessor and Microsystems, 2003.
5. Warmer, J. and J. Kleppe, The Object Constraint Language – Precise Modeling with UML. 1999: Addison-Wesley
6. Kleppe, A. and J. Warmer, Extending OCL to Include Actions. Lecture Notes in Computer Science, 2000. **1939**: p.440-450
7. Cengarle, M. and A. Knapp, Towards OCL/RT. Lecture Notes in Computer Science, 2002. **2391**: p.390-409

Biography

R. D. Hawkins, MSc., Research Associate, Department of Computer Science, University of York, York, YO10 5DD, UK, telephone - +44(0)1904 433385, facsimile - +44(0)1904 432708, e-mail – richard.hawkins@cs.york.ac.uk.

Richard Hawkins has been a Research Associate in the BAE SYSTEMS funded Dependable Computing Systems Centre at the University of York since November 2001. He is researching the use of object oriented techniques for safety critical systems. Before taking up his current role, he attained an MSc in Information Systems from the University of Liverpool and worked as a safety adviser for British Nuclear Fuels since 1997.

J. A. McDermid, PhD., Professor of Software Engineering, Department of Computer Science, University of York, York, YO10 5DD, UK, telephone - +44(0)1904 432726, facsimile - +44(0)1904 432708 e-mail – john.mcdermid@cs.york.ac.uk

John McDermid has been Professor of Software Engineering at the University of York since 1987 where he runs the high integrity systems engineering (HISE) research group. HISE studies a broad range of issues in systems, software and safety engineering, and works closely with the UK aerospace industry. Professor McDermid is the Director of the Rolls Royce funded University Technology Centre (UTC) in Systems and Software Engineering and the BAE SYSTEMS-funded Dependable Computing System Centre (DCSC). He is author or editor of 6 books, and has published about 250 papers.

I. J. Bate, PhD., Senior Research Fellow, Department of Computer Science, University of York, York, YO10 5DD, UK, telephone - +44(0)1904 432786, facsimile - +44(0)1904 432708 e-mail – iain.bate@cs.york.ac.uk

Iain Bate has been working as a Research Associate since 1994 and is now a Senior Research Fellow. His research interests include scheduling and timing analysis, design for safety including architecture trade-off techniques, and the use of optimisation to derive appropriate design solutions.