Abstract

There is increasing interest in the use of Object Oriented (OO) software technology in the development of safety related systems. OO techniques provide considerable potential benefits for system developers such as increased flexibility and re-use. Most current system safety analysis techniques examine the software in a safety critical system as a whole entity. The analyses typically work “top down” from system hazards, and result in a system-specific safety case. If the benefits of OO, especially reuse, are to be realised in safety critical systems, it is imperative that techniques for performing hazard and safety analysis are developed which fit in with the OO development process and life-cycle. This means that, for example, it is necessary to examine components such as classes and objects in such a way that the results can be used to construct analyses of complete systems.

This paper presents a proposal for how hazard and safety analysis of OO systems may be performed. The proposed process starts at the level of OO conceptual designs by investigating faulty behaviours in state transitions of objects and interactions between the objects in a system. Safety contracts for object interactions can then be derived to capture the safety requirements of the system. A simple example is used to illustrate how the techniques can be applied. The paper then goes on to investigate how the techniques may enable safe utilisation of desirable features of OO such as inheritance and reuse. It also discusses how confirmatory analysis of the system design may be performed to ensure the safety requirements defined in the safety contracts are met.

Introduction

Using an OO approach to the design of safety critical systems has become increasingly of interest for a number of reasons. One of the main motivations is that some systems engineers find it useful to think conceptually in terms of objects. It is desirable when addressing a problem to have as tight a correlation as possible between entities in the real world and entities in the model created to represent it. The OO approach provides a good representation mechanism and has the added advantage of greater consistency between different views of the system than the more traditional structured analysis approach. OO also provides a great deal of flexibility. The best example of this flexibility is the ability to reuse objects and components of a system design. Using objects also improves the maintainability of the system due to the stability of an OO system in the presence of changes. The use of OO also has the potential cost saving advantage of allowing the analysis of incremental changes to the system without having to reassess the whole system. In this way the cost of making the change should be proportional to the size of the change, rather than to the size of the system. The above potential advantages together with other desirable features such as improved scalability and support for concurrency mean that there are considerable attractions to using the OO paradigm for system design.

If OO is to be used for safety critical systems then it is imperative that hazard and safety analysis can be successfully performed to ensure the system will function safely. This paper sets out how such analysis may be performed on OO systems designed using the Unified Modelling Language (UML) (ref. 1). The UML is a complete modelling language that has been accepted as the standard by the Object Modelling Group. It is the most widely used method for OO systems and has become the method of choice for most OO developers. For this reason UML is used as the basis of the analysis technique described in this paper.
Challenges of Analysis

Objects within a system are autonomous. That is to say that they are able to perform operations independently without any control from other elements of the system. In a functionally decomposed system it is normal to have a complex subroutine, e.g. a scheduler, that has control over everything else in the system. This is not the case in an OO system. Functionality is shared amongst objects which must collaborate to achieve the system functionality. The objects interact with each other by exchanging messages. The challenge for safety analysis is the need to assess the interactions between the objects.

Another challenge posed by OO systems is that a single structured development process doesn’t really exist. This provides systems developers with the flexibility to use different notations as they require at different stages of development. However, it is much more challenging to ensure traceability of requirements, and completeness and consistency in the system design. What is required are analysis techniques which can allow developers the freedom to make use of the different notations at different stages of development, but still ensure that traceability, completeness and consistency are maintained.

A powerful feature of OO design is that objects may be ‘specified by difference’ using inheritance. If use is to be made of the inheritance mechanism it is necessary to ensure that the safety properties of the system are preserved, or appropriately refined through inheritance. It is desirable to make use of inheritance but it is imperative that this can be done in a safe manner. To do so will require analysis of the system down to the level of individual classes.

Hazards in an OO System

Functionality is realized in an OO system as a sequence of interactions between objects. Each of these interactions links a client object that requests an operation with a supplier object that performs the operation. In Figure 1, object 1 is the client object which requests an operation from object 2, the supplier. The request is made by an interaction, e1, between object 1 and object 2.

![figure 1](image1.png)

Figure 1 – Functionality as a Sequence of Interactions

This interaction is an event that may trigger a change in state of the supplier object. This change in state of object 2 may give rise to an action that generates a further interaction, e2, with another object. In this way the supplier object 2 has become a client. The process is illustrated more fully in Figure 2. The transition from state A to state B of object 1 is triggered by the event e1 (and condition c being met). This transition leads to an action, which is e2, which generates an interaction with object 2. This action triggers a transition from state E to state D in object 2. It can be seen how this process might continue if action a generated a further interaction with another object.

![figure 2](image2.png)

Figure 2 - Interactions Generate State Transitions

Key:

- \( e[c]/a \) = if event e occurs and condition c is met then perform action a
For an OO system, a hazard arises from an undesirable sequence of events (or interactions). Each of the undesirable events in the sequence is either due to a message being sent incorrectly by the client, or the message not being acted on correctly by the supplier. As was discussed above, these interactions can be generated by client object state transitions. Supplier objects respond to the interaction also by state transitions. Therefore faults in these state transitions can give rise to a system hazard. We propose that by defining a safety contract for the interaction between client and supplier objects, it will be possible to ensure that the interaction will not contribute to the hazard.

Defining the System

To illustrate the analysis process, a simple example will be used. The example is a vastly simplified rail crossing which is used to allow a railway line to cross a road. The cars are controlled by means of a road light that can be either red or green and the trains are controlled by a rail signal that can be either open or closed. The car and train are both operated by drivers who observe the relevant signal.

The starting point for the analysis is to define a set of classes to represent the system, by constructing a conceptual class model. This model will be based on the results of the requirements analysis phase of the development process. It is assumed that this phase has occurred and that some Preliminary Hazard Analysis (PHA) has been undertaken, however it is not within the scope of this paper to discuss how this is carried out. The key consideration in constructing the conceptual model is that it should be a representation of the problem, rather than a model of a potential solution. This means that the aim should be to identify the high level concepts within the system and the relationships between them.

For the rail crossing example, the conceptual model is shown in figure 3. There is an obvious system hazard that should be identified at PHI, that the car and the train are both in the crossing at the same time.

A set of use-cases is constructed at the requirements analysis phase to represent the functionality required from the system. Collaboration diagrams are developed for the use cases relevant to the occurrence of the system hazard. The collaboration diagrams are used to represent the dynamic behaviour of the objects required to meet the use case functionality by showing the interactions between the objects. The use case ‘train approaching crossing’ is relevant to the system hazard, see figure 4.

It is necessary at this point to confirm that under correct behaviour, the system doesn’t allow the occurrence of the system hazard. That is, if the objects in the system correctly generate the intended events, are in the correct state, and assign the correct value to variables, then the system is safe. In effect this is a check to see that the system isn’t hazardous as designed. It is fairly easy in this example to confirm by inspection that the system design will not allow the car and the train to be in the crossing at the same time. When a more complex system is being modelled it may not be so easy to perform this confirmation, in which case a model-checking tool may be required to verify the design is safe.
Having confirmed that the concept is safe when fault-free, the faulty behaviour of the system must be examined. The collaboration diagram can be used to identify undesirable events which can bring about the system hazard. An effective way of achieving this is to use the collaboration diagram to construct a fault tree. Fault tree analysis is a well-established graphical analysis technique which works backwards from a ‘top event’ directly related to the identified hazard to determine its cause (ref. 2). This is done by combining intermediate events related to the top event using logical operations. The fault tree for the rail crossing can be seen in figure 5. The top event is the car and train both being in the crossing. The collaboration diagram above defines the events that should occur in a system that avoids this top event. The collaboration diagram is therefore also useful in determining the events that could contribute to this top event.

Figure 5 – Fault Tree for Rail Crossing
It is necessary to identify which elements of the collaboration diagram relate to each event in the fault tree. In this example each of the undeveloped events, represented by the diamonds, relates to an object in the system. For example the undeveloped event ‘controller malfunction’ relates to the crossing_control object, and ‘car driver fails to react to red light’ relates to a failure in the car driver “object”. Other events in the fault tree can be seen to relate to interactions in the collaboration diagram. For example ‘light not set correctly’ relates to some failure in the setLight() interaction, and ‘driver receives wrong information from light’ relates to the updateColour() interaction.

For each of the objects identified in the fault tree as potentially contributing to the hazard, whether as a supplier, a client, or both, a state chart must be developed to model the behaviour of that object. For an example we choose to analyse the car_driver object. The state chart for this object can be seen in figure 6.

![State Chart for the car_driver Object](image)

From the state chart, the state transitions which relate to the relevant interactions identified in the fault tree must be identified. For client objects these state transitions will be those responsible for generating the event. For supplier objects, it will be the state transitions reliant on the event. This particular object is a client of the drive() interaction, which relates to the ‘car fails to stop as required’ element in the fault tree, and is also a supplier of the updateColour() interaction, which is responsible for the ‘driver receives wrong information from light’ element of the fault tree. For this object all the transitions are of interest for the object both as a client generating the drive() interaction, and as a supplier responding to the updateColour() interaction. For more complex state charts there may be a large number of states and state transitions. It is important that only the relevant transitions are picked out for further analysis. For the identified transitions the potential faulty behaviours must next be identified and simulated.

### Identifying Unsafe Behaviours

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 which 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. This idea was initially developed by Gorski and Nowicki (ref. 3). The guidewords used are similar to those used in the functional failure analysis (FFA) technique, namely omission, commission and value. These guidewords are applied to each element of each of the relevant transitions. In order to be able to simulate these faulty behaviours, extra transitions must be added to represent them in the state chart. The application of the guidewords to each transition element is shown below along with the extra transitions necessary to simulate the faulty behaviour.

- **event – omission** – ‘is ignored’ – add self-transition \(e/c\)
- **commission** – ‘spuriously generated’ – add parallel transition \(!e/c/a\)

- **condition – omission** – ‘is ignored’ - add self-transition \(e/c\)
- **commission** – ‘taken as true when false’ – add parallel transition \(e/!c/a\)
As can be seen some of the extra transitions used to simulate the faulty behaviour occur more than once. There are in fact just five distinct transitions identified which are as follows:

1 - \( e[c] \) self-transition – event or condition is ignored
2 - \( \text{not } e[c]/a \) – event spuriously generated or action performed without initiating event
3 - \( e[\text{not } c]/a \) – 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 relevant transitions, these five extra transitions are added to the diagram to simulate their faulty behaviour. To illustrate this, the result of applying the five transitions to the car_driver state chart is shown in figure 7 below.

![Figure 7 – Faulty State Transitions for car_driver Object](image)

The faulty behaviours of the object have now been identified. Not all of these behaviours would lead to an unsafe condition. To identify the unsafe behaviours we must pick out the faulty behaviours which will contribute to the system hazard. In the above example there are three such behaviours:

1C – condition is ignored – the driver does not see the red signal.
3B – condition taken as true when false – the driver sees the signal but thinks it is green (colour blind?)
5C – wrong action performed – the driver sees the signal and knows it is red but continues anyway.

What is common to these three transitions is that they all lead to the car continuing when the light is at red.

### Constructing Safety Contracts and Requirements

The identified faulty behaviour of the objects can be used to construct safety contracts for the interactions. As the car driver object acts as both a supplier and a client it must first be identified which of the unsafe behaviours relates to which role. For the first of the three unsafe behaviours the interaction
(updateColour()) has occurred but not triggered the required response. This indicates that this behaviour relates to the car driver as a supplier. For the second and third behaviours the transition has occurred without the correct interaction (drive()) being triggered. This information can now be used in constructing the safety contracts for the updateColour() and drive() interactions.

<table>
<thead>
<tr>
<th>Drive(d)</th>
<th>SetLight(col)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Client:</strong> car_driver</td>
<td><strong>Client:</strong> crossing_control</td>
</tr>
<tr>
<td><strong>Supplier:</strong> car</td>
<td><strong>Supplier:</strong> road_light</td>
</tr>
</tbody>
</table>
| **Requires:** IF [colour=red & in in_front_of ] / d=stop | **Requires:** IF setSignal(open)/col=red
IF NOT train_left /col<>green |
| **Guarantees:** IF d=stop / in waiting | **Guarantees:** IF col=red /turn_red
IF col<>green /NOT turn green |

Figure 8 – Safety Contracts for Interactions

A safety contract will be defined for an interaction by defining four things. Firstly the name of the interaction covered by the contract is specified. The object or objects which are client and supplier are also stated. The final two elements of the contract are the requirements and the guarantees. Requirements define the behaviour which is required from a client object to ensure the interaction doesn’t contribute to the system hazard. Guarantees define the behaviour that the supplier object must guarantee to ensure the interaction doesn’t contribute to the system hazard. Figure 8 defines the safety contract for the drive() interaction. The unsafe behaviour identified above for the car driver object as a client is used to define the required behaviour. The contract was completed by carrying out the same analysis on the car object, as the supplier, to generate the guarantee behaviour. Figure 8 also contains as a further example the safety contract for the setLight() interaction. A similar contract is formed for all the interactions identified earlier in the fault tree as potentially contributing to the system hazard. The unsafe behaviours identified for car driver as a supplier are used to form the guarantee conditions of the updateColour() interaction.

Figure 9 – Conceptual Model with Derived Safety Requirements
Objects in the system may be clients and suppliers in many interactions in the system. The objects must meet the requirements of the safety contracts of all the interactions for which it is a client and must guarantee the behaviour defined by the safety contract of all the interactions for which it is a supplier. This places requirements on the classes in the conceptual model. These requirements define the behaviour of the class which is critical to the safety of the system. If a class fails to meet any of these requirements then a safety contract will have been broken and an event identified as contributing to the system hazard could occur. These derived safety requirements can be added to the conceptual model as shown in figure 9.

**Developing a Process**

The analysis techniques used must fit in as part of an overall development process. As mentioned earlier, it has been assumed that requirements analysis has occurred prior to the conceptual model being constructed, and that PHI has been undertaken. The analysis described in this paper then concentrates on this high level design of the system. If it can be shown that further refinement of the design is a valid representation of the design in the conceptual model, and meets the safety requirements laid out in that model, then hazard analysis down to the implementation level is not necessary. The key thing is being able to show that the detailed design is at least as safe as the system analysed. This will require some form of confirmatory analysis. The process is represented in figure 10.

**Figure 10 – Analysis as Part of Development Process**

The thick dashed line is a simple representation of the software development ‘V-model’. The thick solid line represents the analysis process. As can be seen, the analysis process runs concurrently with the software development process until the conceptual model analysis described in this paper (contained within the dashed rectangle in the diagram) has been completed. At this time the more detailed design of the system will continue without further safety analysis. Where the analysis and development processes next meet, confirmatory analysis of the design against the earlier analysis is performed.

**Utilising Safety Requirements**

When the more detailed design of the system is performed, the safety requirements identified in the conceptual model must still be honoured. In practice, a single concept in the conceptual model may be implemented as a number of different classes in the more detailed design. For example, crossing control is a single concept in the conceptual model of the rail crossing. When the system is designed in more detail, this concept may be implemented as a number of different classes. This does not cause a problem so long
as the behaviour exhibited by the whole collection of objects making up the crossing control meets the safety requirements of the crossing control concept. These classes are not adding any new concepts to the conceptual model, merely implementing an existing concept. Therefore interactions between these classes do not need analysis as long as no new interactions with other concepts beyond those already existing in the conceptual model are introduced. If additional concepts, or additional interactions between concepts are introduced to the conceptual model, then their contribution to the system level hazard needs to be considered. This may involve developing safety contracts for the safety related interactions introduced.

It may be possible to define safety requirements for a super class which can be inherited by child classes. The inherited requirements are only valid for the same interactions as those of the super class. If a child class has extra interactions with other concepts in the conceptual model, then the inherited requirements will not be sufficient and additional analysis of the extra interactions would be required. As an example of how safety requirements may be inherited a super class ‘driver’ may be constructed. It can be noted in figure 9 that the safety requirements for car_driver and train_driver are identical except that they are specific to the particular vehicle, for example road light or rail signal and car or train. By generalising the safety requirements the safety contract for driver can be constructed as shown in figure 11.

![Figure 11 – Inheriting Safety Contracts](image)

The safety requirements of the super class could be inherited in the same way as attributes or operations would be. This process would allow, for example a lorry_driver or cyclist class to be added to the system and inherit the safety requirements of driver. In the example system, another potential application of this would be to create a super class of road_light and rail_signal called control_signal. Again this could allow the safe introduction of additional types of signal to the system.

**Future Research**

There remains much research to be done on performing hazard and safety analysis of OO systems. This paper presents just one possible technique which may form part of an analysis process. This technique must be developed further if it is to be used successfully on more complex systems. In particular, timing and sequencing issues have not been dealt with in this paper. There will clearly be safety issues associated with such properties of a system and these must be incorporated into the analysis technique. It is felt that sequence charts, which have not formed part of the analysis presented here, may provide the high-level timing and sequencing information that is required for this, so they will be investigated further. The mechanism for identifying unsafe behaviours from the many faulty behaviours will also need to be developed. In the trivial example presented here it is fairly easy to identify which of the faulty behaviours are unsafe, however in a system with more complexity, this may be a more difficult process.

The analysis technique discussed in this paper contains both inductive and deductive elements, but is mainly a predictive technique. However techniques for performing confirmatory analysis need to be developed. An important part of this will be ensuring there is a consistency between different UML views of the system. This is imperative in ensuring that the detailed design is a true representation of the conceptual model.

Also, the analysis assumes that failures in objects in the system are independent, however in reality this will rarely be the case. Common cause analysis techniques could be used to identify non-independence of failures within the system and their applicability to the process needs to be investigated. Sneak analysis
(ref. 4) is a technique which was originally developed for use in identifying unintended paths (sneak paths) through electrical circuits. There may be an application for a sneak analysis style technique in identifying sneak paths through, for example, a collaboration diagram. Again this requires further research.

The representation of safety contracts and safety requirements in this example is done using a combination of standard logical expressions, UML and Object Constraint Language (OCL) (ref. 5) syntax. The emphasis was on presenting the information clearly to the reader. If the technique is to be used to specify behavioural requirements to designers and engineers then a more formal representation of these properties must be developed. All of the future research work discussed here will require a more complex and realistic example to test the techniques being developed. It is expected that an industrial case study, probably within the aerospace domain, will be used for this purpose.

Conclusions

This paper has presented a proposal for how hazard and safety analysis of OO systems may be performed. This has been demonstrated using an example system. The paper demonstrates that thorough analysis can be performed on systems designed using OO techniques. It also demonstrates that there is potential, with further research, to use OO successfully in safety critical applications. The benefits of being able to do this have been briefly explored. The limitations of the technique at its current state of development have been acknowledged and future research to develop the technique further has been suggested.

References


Biography

R. D. Hawkins, MSc., Research Associate, Department of Computer Science, University of York, York, YO10 5DD, UK, telephone - +44(0)1904 433385, 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. Before taking up his current role, he attained an MSc in Information Systems from the University of Liverpool and had 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.