An Approach to Designing Safety Critical Systems using the Unified Modelling Language

Richard Hawkins, Ian Toyn, Iain Bate

Department of Computer Science
The University of York
Heslington, York
YO10 5DD, UK
{richard.hawkins, ian.toyn, iain.bate}@cs.york.ac.uk

Abstract. In this paper an approach to using the UML for developing safety critical systems is presented. We describe how safety analysis may be performed on a UML system model and how this analysis can derive safety requirements for classes in the system. We show how these requirements can be expressed in the form of *safety contracts* using the OCL. This makes it possible to reason about the safety of individual elements of the UML model and thus makes it easier to safely change the UML design, as well as facilitating maintenance and reuse of classes or components in the system. A tool is also described which has been developed to automate some aspects of this analysis.

1 Introduction

There is increasing interest in the use of an Object Oriented (OO) approach for developing 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. These benefits could bring potentially large savings in terms of time and cost for developers of safety critical systems. However they also raise specific challenges which must be addressed if the full advantage of these OO features are to be realised for safety critical systems. To realise these benefits requires an ability to reason about the safety of individual classes or components in a system. This requires firstly that safety and hazard analysis be successfully performed on OO designs.

The Unified Modelling Language (UML) has become a de-facto standard for modelling OO systems and is widely used throughout industry. In this paper we initially look at performing hazard analysis of UML models. Moreover, however, most existing safety analysis techniques tend to decompose hazards in a functional manner. It is therefore difficult to reason about the safety of individual classes or components. If the benefits discussed earlier are to be realised, these existing techniques must be adapted such that the required results are obtained. Therefore we go on to look at how safety requirements may be captured in the form of safety contracts. These safety contracts can be specified using the Object Constraint

Language (OCL) and incorporated into the UML model of the system. This could contribute towards the development of a safety critical profile for UML. We show how the safety contracts can be generated through analysis of different aspects of the UML model, covering functional, timing and value behaviours.

2 System Safety Analysis

Leveson provides us with the following definitions [1]. *Safety* is the freedom from accidents or loss. Safety critical systems are systems which have a direct influence on the safety of their users and the public. A *hazard* is a state or set of conditions of a system that, together with other conditions in the environment of the system, will lead inevitably to an accident. The primary concern of system safety analysis is the management of hazards: their identification, evaluation, elimination, and control through analysis, design and management procedures. One aspect that distinguishes system safety from other approaches to safety is its primary emphasis on the early identification and classification of hazards so that corrective action can be taken to eliminate or minimize those hazards as part of the design process. The system safety process culminates in a safety case being produced that gathers all the necessary evidence to justify the system is safe.

The system safety analysis process can be basically split into the following steps:

- Hazard identification – This step identifies the potential hazards in the proposed system.
- Risk assessment – This examines each of the identified hazards to determine how much of a threat they pose. This assists in deciding the steps required to reduce the risks to acceptable levels. Many initial safety requirements are set at this stage.
- Preliminary system safety assessment (PSSA) – This phase is concerned with ensuring that a proposed design can meet its safety requirements and also with refining these safety requirements as necessary
- System safety assessment – This stage is concerned with producing the evidence that demonstrates the safety requirements have been met by the implementation.
- Safety case delivery – This involves producing a comprehensive and defensible argument that the system is safe to use in a given context. The analysis performed as part of the PSSA stage will form part of this argument.

In this paper we are primarily concerned with the PSSA. This is very closely linked with design activities. A key part of this is the identification of specific derived safety requirements to guide the detailed design of the system. The majority of the safety analysis techniques used in PSSA are deductive. This means that they investigate possible causes of a specific hazard or condition, starting from system level hazards and requirements identified during the hazard identification phase. The majority of techniques used for PSSA tend to 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 than in an OO system 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 classes and objects. The functionality of the system is realised by many objects collaborating through message passing to achieve a system function. Therefore a functional failure will not map easily onto a single element of the design. In theory it would be possible to put all functionality into just one class and to apply standard safety analysis techniques. In doing this however, all advantages of adopting an OO approach would be lost. It is important therefore, for OO systems and indeed for UML system models that existing techniques can be adapted such that failures can be allocated between classes.

Our approach achieves this by allocating derived safety requirements to individual classes and controlling the interactions between classes through the use of contracts. This allows the impact of changes to be understood. An appropriate allocation of constraints would also allow better support for change in that change would be contained within a class or within a small hierarchy of classes. Other work looks at this in more detail.

3 Safety Analysis for UML

There have been attempts to adapt safety analysis techniques to apply them to UML. In [2], Lano et al examine how Hazard and Operability Studies (HAZOP) can be adapted to UML. HAZOP is a predictive safety analysis technique which uses a set of guidewords to consider the behaviour of ‘flows’ between system components. Lano et al take the standard guidewords for HAZOP defined in Defence Standard 00-58 [3] and reinterpret them such that they are applicable to different views of the UML model. For example, when considering state transition diagrams, the ‘flow’ is taken to be the transition and the guidewords are interpreted accordingly. Similarly, for class diagrams ‘flows’ are taken to be relationships. This is a very useful approach to analysing failures, however it would demand that the technique be applied to every class, attribute, state transition and interaction in the system to be effective. Even for a fairly small system this could be prohibitively time consuming.

Nowicki and Gorski have also developed analysis based largely around state charts. In [4], three methods are introduced. The first method, detailed in [5] centres around the development of a hazard model, which explicitly models safe and unsafe states and the transitions between these states. Reachability analysis can then be used to check if the hazardous state is reachable. This method assumes that the system and environment are reliable in the sense that they behave as specified. The second method [6] accepts that this assumption isn’t always valid as objects are exposed to random failures and the environment can violate assumptions made about it. This method provides a set of templates of faulty behaviour, which are deviations from the normal behaviour of the object. The templates consider possible faults in transitions and are applied to the state chart of the object under consideration. This generates a state chart for an ‘unreliable object’. Reachability analysis is then performed to see if a hazard can occur for the unreliable object. Whereas the first two methods were concerned with analysing the system design, the final method aims to strengthen the safety guarantees of the system by enriching the system with a ‘safety monitor’ object. This is of less interest here. It is the second method which is perhaps most useful. There are certain weaknesses, particularly in identifying which objects to apply the templates to. Again, applying them to all objects in the system would be potentially very time consuming. It is also unclear how hazardous states are correctly identified.

In the remainder of this section we take some of the ideas discussed above and incorporate other techniques such as Fault Tree Analysis (FTA) to produce a more focussed approach to analysing UML models such that safety requirements can be derived for classes in the system in the form of safety contracts. Safety contracts constrain the design of the interactions which occur between objects, and hence can ensure system behaviour is safe. 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 which 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.

3.1 Functional Aspects

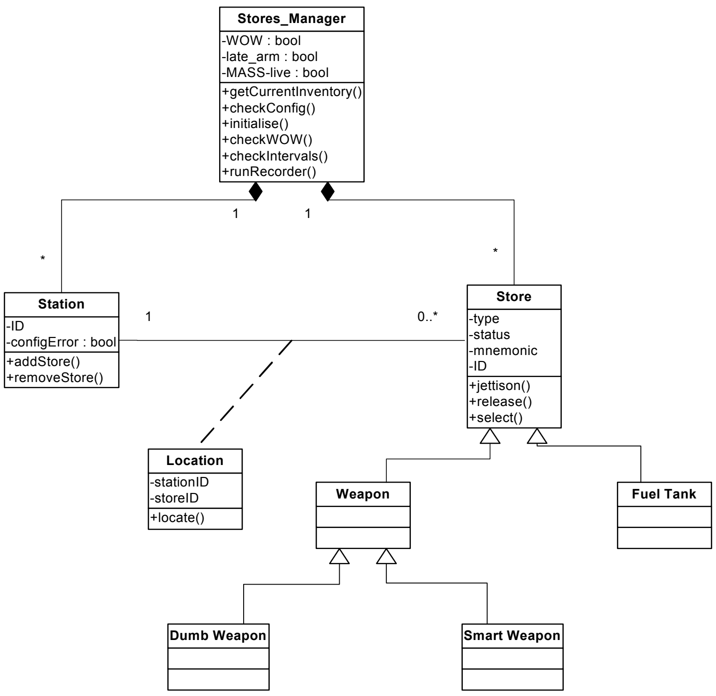
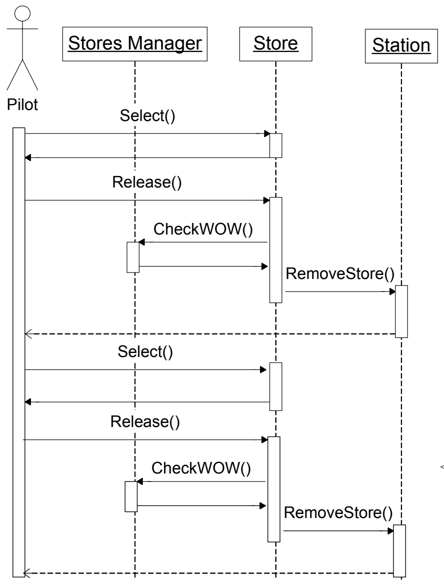


Fig. 1 – Class Diagram for Part of a Stores Management System

To illustrate the analysis we use an example of a highly simplified aircraft Stores Management System (SMS). In the context of an aircraft, a store could be such things as a weapon or a fuel tank. These are connected to the aircraft via stations on the wings. The UML class diagram for this system is shown in figure 1. The initial hazard identification process performed on the SMS identified a number of general 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



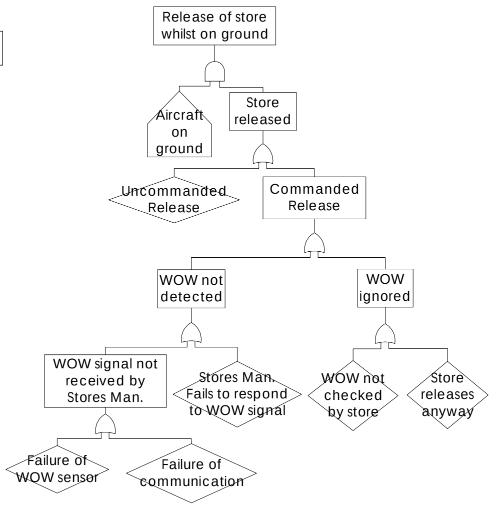


Fig. 2. UML Sequence Diagram for a Normal Operation Scenario.

Fig. 3. Simplified Fault Tree for the Release of Store Whilst on the Ground

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 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 2. A fault tree is constructed using system information collected from the UML diagrams and domain knowledge. Fault Tree Analysis (FTA) represents graphically the combination of events and conditions which contribute to a single undesirable event. Starting from the top event, the immediate causes of that event are identified. The manner in which these causes contribute to the event is expressed with the use of logic gates (basically AND and OR). This continues down the tree until the tree consists entirely of basic events, i.e. events which cannot be decomposed further, or until the desired level of detail has been reached. The fault tree in figure 3 shows the faults that can occur to bring about the top event "Release of store whilst on ground". It should be noted that WOW is 'Weight on Wheels', used to indicate when the aircraft is on the ground.

It is possible to relate leaf nodes (undeveloped failure events) in the fault tree, represented by diamonds, 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. Not all leaf nodes in the fault tree will relate to classes in the system design. Of the six leaf nodes in figure 3, three can be related to classes in the system design. For example 'Failure of WOW sensor' cannot be attributed to a class, since the WOW sensor is not part of the Stores Management System. For the three leaf nodes identified as related to the SMS' classes, 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 1.

Table 1. Table of hazardous class behaviour

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

In this table, the hazardous event field is the event identified in the fault tree. The class field is the class with which that event is associated. The interaction is the operation identified from the interaction diagram as corresponding to this event. The role is that played by the class in that interaction (either client or supplier of that operation call), and the hazardous class behaviour is that which would be exhibited by an instance of the class to bring about the hazardous event.

3.1.1 Analysing State Charts

We have now derived hazardous conditions for classes in the system. It is therefore possible to start constructing safety requirements and safety contracts for these classes. However to identify more detailed derived safety requirements it is necessary to understand how the class may behave such that these conditions can occur. This can be done by studying the state charts of these classes. 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 4. A store may be jettisoned or released, jettison is a special case of release when a store is in a 'safe' state (e.g. dropping an unarmed weapon). The hazardous class behaviour for the Store class taken from table 1 can be summarised as:

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

Now that hazardous states have been defined it is possible to apply the ideas of Gorski and Nowicki discussed earlier [6]. 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 behaviour defined above. Checking that a proposed design does not exhibit hazardous behaviour can be achieved using a reachability analysis tool. 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 due to mistakes in implementation, must also be investigated. To do this we mutate the transitions in the state chart [6]. 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 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 (omission)

2 - not e[c] / a – event spuriously generated or action performed without initiating event (commission)

3 - e[not c] / a – condition taken as true when false (value)

4 - e[c] – action is ignored (omission)

5 - e[c] / b (where b is an action other than a of the initiator object) – wrong action performed (value)

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 5. The transitions between the unselected and selected states have not been included as they are not relevant to the hazardous behaviour we are interested in. 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.

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*

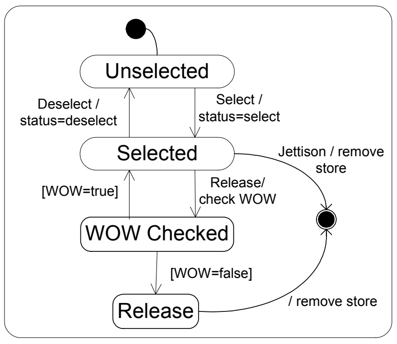


Fig. 4. State Chart for Store Class

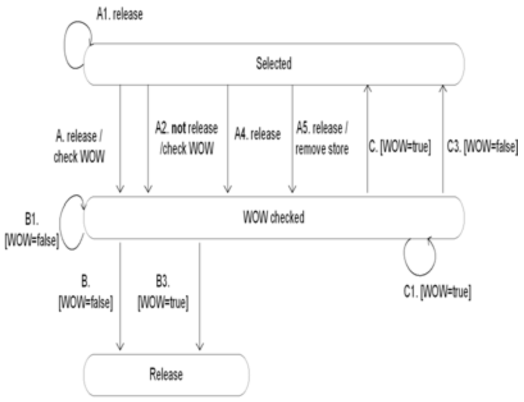


Fig. 5. Mutated State Chart Showing Faulty Transitions

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 and is discussed in section 5. 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. In defining a safety contract we are only interested however in constraining the hazardous behaviour and not all correct 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. We look at constructing these contracts in section 4.

3.2 Timing 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 [7]. 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 2. 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 table 2.

Table 2 – The effects of timeliness of tasks on system hazards

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 release 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**

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.

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 have as much tolerance as possible whilst still constraining the task sufficiently to ensure it is safe. This will allow maximum flexibility to the implemented system. In [8] 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 is based on decomposing the systems objectives to a number of design choices and assessment criteria. Each design choice can be evaluated using a combination of static analysis and simulation. Heuristic search techniques can then be 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. We have found activity diagrams to be particularly suited to this application as they emphasize the sequential and concurrent nature of the tasks in a scenario. However, it is acknowledged that sequence diagrams could also potentially be adapted for this purpose. 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. These scenarios could also necessitate additional timing requirements, which will form part of the safety contract (see section 4).

3.3 Value Aspects

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. If 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. It is therefore important for safety critical systems that attributes of classes are declared as private such that they may be constrained in this manner.

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 constraints it is possible to assure the store ID being passed is correct.

The nature of constraints 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

4 Specifying Safety Contracts

Once the safety requirements for the classes in the system have been derived it is necessary to specify them in a useful and meaningful way. There are a myriad of techniques such as Douglass' real-time annotations [9] suitable for specifying constraints in UML. In this work we have chosen to use OCL [10], which is a constraint language compatible with UML. The pre- and post-condition constraints from OCL can be used as the basis for safety contracts on operations. 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 example in section 2, 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 [11] 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** [12]. 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()
```

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 [13]. 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 + 100 <= Time.now
          post: WOW=false
          and
          Time.now <= Time.now@pre + 50
          and
          Stores_Manager ^ checkWOW()
```

By using OCL to specify contracts in this way the safety requirements on each class in the system design are explicit. For all our case studies, OCL with the extensions described has proved suitably expressive. If necessary however, other constraint languages or extensions could be considered. To ensure it will not impact on the safety of the system, the class must meet the set of safety requirements consisting of all preconditions 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. Specifying contracts in this way also makes it easier to deal safely with maintenance, change, and reuse. This is discussed more thoroughly in [14].

5 Tool Support

Our tool support, as developed to date addresses the analysis discussed in section 3.1.1. Tool support for this part of the analysis is particularly important given its potential complexity for real systems. The tools developed provide assistance in a number of ways. They can identify sequences of transitions that could lead to an identified hazard. The tools are also capable of generating mutations of transitions which may arise due to mistakes in implementation. It is possible to perform these in combination to identify which mistakes in implementation could lead to an identified hazard. The tools can also be used for test data generation by identifying what input values would trigger a transition, and what output values would result from that transition and inputs. This allows us to create test cases for any of the hazardous potential mistakes in our implementation. Before any of this can be done however it is first necessary to check that the state chart under consideration uses only notations that are analysable by the tools, and to formalise the state chart ready for analysis.

5.1 Well-formedness and Formalisation

The tool first checks that the state chart provided is analysable, that is that the state chart is well-formed. If this is the case then the state chart is translated into an ISO standard Z representation [15]. Both of these steps are fully automated. Once the tool has been initiated with the state chart design no user intervention is necessary. The rest of the toolset works from the resulting Z representation. This should ensure that the analysis is less dependant on any particular state chart dialect. We currently cope with Statemate and Stateflow. Each of the transitions in the state chart is represented by a Z schema. The inputs and outputs of the state charts become inputs and outputs of the schema. The labels on transitions become predicates over these inputs and outputs.

5.2 Mutant Generation

In section 3.1.1 it was shown how extra transitions can be added to the state chart to simulate faulty behaviour. These extra transitions are referred to in the toolset as mutant transitions. The five basic cases in 3.1.1 are the basis for the mutant generation by the tool.

1 - e[c] self-transition – This involves changing the destination state to be the same as the source state and dropping the action.

2 - not e[c] / a and 3 - e[not c] / a – These are generalised to negate any conjunct of the e[c] trigger, and also to negate the entire trigger. Where the trigger involves ordering relations, each can be mutated to use a different relational operator, e.g. < becomes <=.

4 - e[c] – This involves deleting the action. If the action is a sequential composition, further mutants are generated by deleting one of those actions.

5 - e[c] / b – This involves changing the actions. Where an action is the assignment of a Boolean, the Boolean assignment can be negated. Where an action is an

assignment of a number, the number assigned can be incremented/decremented, and any arithmetic operator can also be changed, e.g. + to –.

The tools generate all mutants that they can, for all transitions. The user merely initiates the mutant generation. It should be noted that using the tools allows many more mutants to be considered than would otherwise be feasible.

5.3 Hazard Detection

Section 3.1 illustrated the identification of hazards for a class in the system. In section 3.1.1 these were shown in a notation that is basically Z predicates, but which need declarations of the names to be legal Z, these were:

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

By writing them in a Z schema with inclusion of the state chart's inputs and outputs, the identified hazards become available to the tools. This small and relatively simple step currently has to be done manually, but since it involves manipulation of Z could be automated. Hazard detection involves conjecturing whether a composition of transitions implies the hazard. The hazard might be implied never, sometimes or always. This can be done for the original state chart and for the mutant transitions. When composing only original transitions, *sometimes* is sufficient for concern. When composing transitions involving mutants, it may be preferable to focus on the *always* cases. The user chooses the maximum length of compositions to be considered, then the tools perform automatically. The cost increases rapidly with the number of alternative transitions exiting each state. Generating a large number of mutants can also have a devastating effect on the cost of hazard detection. It would be possible to allow the user to limit the number of mutants generated by the tool in order to ensure hazard detection remained practical. This could be based on the complexity of the initial state chart and the severity of the resulting hazard. Further work may look at this.

5.4 Test Data Generation

Test data generation for a transition amounts to finding a binding that exists in the schema. This determines input and output values simultaneously. The solutions are written out in a form suitable as input to a test harness tool. The user has only to choose the transition for which to generate test data. This allows tests to be constructed to check for the existence in the implemented design of any of the hazardous mutant transitions identified previously by the toolset.

6 Conclusions and Future Work

In this paper we have outlined an approach to developing safety critical systems using UML. This is based on analysis of the UML system model. We have described with a simple example how this analysis may be carried out. We have also described a tool for automating some aspects of this analysis. This analysis derives safety requirements for classes in the system. We have shown how these requirements can be expressed in the form of safety contracts using OCL. This approach makes it possible to reason about the safety of individual elements of the UML model and thus makes it easier to safely change the UML design, as well as facilitating maintenance and reuse of classes or components in the system.

Future work will focus on verification that a design will meet derived safety requirements arising from safety contracts, and deal with violation of conditions. We will also look at supporting traceability, particularly with a changing or evolving UML model. It will also be necessary to define safety arguments in support of this approach such that UML may be successfully adopted into safety critical domains.

The work presented in this paper was funded by BAE Systems through the DCSC, and by the EPSRC through the MATISSE project. A great deal of work on developing the toolset was carried out by Stuart Bass, an intern student at the University of York.

References

1. Leveson, N., Safeware - System Safety and Computers. 1995: Addison-Wesley.
2. Lano, K., D. Clark, and K. Androutsopoulos, Safety and Security Analysis of Object Oriented Models. Lecture Notes in Computer Science, 2002. **2434**: p. 82 - 93.
3. MoD, Defence Standard 00-58: Hazop Studies on Systems Containing Programmable Electronics. 1996: HMSO.
4. Nowicki, B. and J. Gorski, Object Oriented Safety Analysis of an Extra High Voltage Substation Bay. Lecture Notes in Computer Science, 1998. **1516**: p. 306-315.
5. Gorski, J. and B. Nowicki, Safety Analysis Based on Object-Oriented Modelling of Critical Systems. Proc. SAFECOMP '96, 1996: p. 46 - 60.
6. Gorski, J. and B. Nowicki, Object Oriented Approach to Safety Analysis. Proc. ENCRESS '95, 1995: p. 338-350.
7. Douglass, B.P., Real-Time UML - Developing Efficient Objects for Embedded Systems. 1998: Addison-Wesley.
8. Bate, I., J. McDermid, and P. Nightingale, Establishing Timing Requirements for Control Loops in Real-Time Systems. Journal of Microprocessors and Microsystems, 2003. **27**(4): p. 159 - 169.
9. Douglass, B.P., Doing Hard Time - Developing Real-Time Systems with UML, Objects, Frameworks, And Patterns. 1999: Addison-Wesley.
10. OMG, Object Constraint Language Specification, in Unified Modeling Language Specification version 1.4. 2001, Object Management Group.
11. Kleppe, A. and J. Warmer, Extending OCL to Include Actions. Lecture Notes in Computer Science, 2000. **1939**: p. 440-450.
12. Warmer, J., et al., Response to the UML 2.0 OCL RfP - Revised Submission, Version 1.6. 2003, OMG.
13. Cengarle, M. and A. Knapp, Towards OCL/RT. Lecture Notes in Computer Science, 2002. **2391**: p. 390-409.
14. Hawkins, R.D. and J. McDermid, Developing Safety Contracts for OO Systems. Proc. 21st International System Safety Conference, 2003.
15. ISO, Information Technology - Z formal Specification Notation - Syntax, Type System and Semantics. First Edition ed. 2002: ISO / IEC