Developing Successful Modular Arguments for Object Oriented Systems

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

S.A. Bates; University of York; York, UK

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

Keywords: Safety Cases, Modular Safety Arguments, OO, Safety Case Architectures

Abstract

In previous work, we have independently reasoned about developing "safe" object-oriented (OO) systems, and constructing modular safety arguments. One of the challenges currently under consideration is how to successfully certify safety-critical OO systems developed using this approach. It was concluded that in order to maintain the desirable properties of OO systems such as re-use and inheritance, the traditional monolithic approach to safety argument construction was too inflexible. A modular argument approach is instead proposed.

In this paper we explore different modular safety argument architectures. A preferred modular structure is identified from a number of alternatives based on the ability to support change and reuse of an OO design. The arguments and evidence required for developing each module in the preferred structure is then presented.

Introduction

There is increasing interest in the use of OO techniques in safety critical domains such that the benefits experienced in other domains may be utilised. The benefits have been claimed to include increased changeability due to abstraction and encapsulation, and increased facilities for reuse through generalisation and inheritance (ref. 1). It is necessary, if this is to be done successfully, it can be shown that OO can be utilised without unduly affecting the safety of the system. It has been shown previously (ref. 2) how a safety contract approach to developing OO systems can reduce the amount of re-analysis required when changes occur. If the value of this is to be fully realised, however, re-certification effort resulting from changes must also be reduced. It has been proposed that this can be done by adopting a two-stage approach involving modular and incremental certification.

The production of a safety case is an important part of the certification process for safety related systems. A key part of this is providing a clear and defensible safety argument that a system is acceptably safe to operate within a particular context. Existing practice is to certify safety-related systems in a monolithic manner. That is to say that the whole system is certified as one entity. This means that whenever re-certification needs to be carried out, for example after a change to the system design, that re-certification must be conducted on the whole system rather than just on the part of the system that has changed. It is also the case that a monolithic safety case must be produced to justify that the system is safe to commission and then another produced after each change. This means that the costs associated with certifying a system can be very high. We believe that this approach is not desirable for an OO system due to its inflexibility. In this paper we propose a better argument structure and describe how the required evidence may be generated.

In the next section we discuss some issues relating to the use of OO software in safety related applications. This involves the identification of change and reuse scenarios typical of an OO system. Section 3 introduces the concept of a safety case architecture, before exploring some alternative modular architectures for the argument for an OO system. These structures are assessed for their ability to deal with the change and reuse scenarios. Based on this assessment the most desirable argument structure is suggested. Section 7 shows how these argument modules may be developed for an OO system, and how the evidence required to validate these arguments can be obtained through the contract-based analysis process discussed previously.

OO Software for Safety Related Applications

Interest in the use of OO techniques for safety-related applications is increasing as its use in other domains becomes more prevalent. One of the biggest potential benefits of moving to an OO approach is the improved maintainability

gained from the encapsulation, high cohesion and low coupling of OO systems. Inheritance (where classes may inherit the attributes and operations of a parent class) also provides an efficient way of reusing existing elements. Some typical change and reuse scenarios for an OO system are described here.

There are a number of changes that could be considered for an OO design. The most obvious change that may occur is to the design of a class in the system. This means that the implementation of the class changes through, for example, new or enhanced functionality. This is the simplest form of change to deal with if the interactions with other classes are unaffected.

More complex change scenarios occur if they result in changes to the interactions between instances of classes in the system. This could occur if, for example, a new class were added to the system design. This new class may interact in new ways with other classes in the system. The impact of these new interactions on the safety of the system needs to be considered. This will involve carrying out analysis to check that the interactions will not cause behaviour which could contribute to an identified system hazard.

Due to the encapsulation inherent in an OO system, there is an increased potential for reusing elements in another system. This is because encapsulation ensures that the data relating to an object, and the operations that manipulate that data are combined in a single entity, thus greatly reducing the interdependency between objects. We will consider a reuse scenario where we wish to use a class from an existing system design as part of the design for another similar system. The advantage of this is that it saves on the expense involved in designing that part of the system from scratch. For a safety related system, however, it must be shown that the class is being reused in a safe manner, which means producing a safety argument for the changed system. To get as much advantage from the reuse as possible, it is desirable to reuse as much of the existing safety argument as possible in the safety argument of the new system.

For safety-related systems it must of course be shown that this kind of change and reuse can be achieved in a safe manner. The key to ensuring that change and reuse can be achieved safely is to be able to reason about, or in some way ‘encapsulate’, information about how individual elements of a system design must behave to ensure that when they collaborate together they do not contribute to system level hazards, and therefore to the safety of the system. If this encapsulation can not be achieved then the benefits strived for in adopting an OO approach will be lost, as the safety process would not be aligned with the design.

Unfortunately it is difficult with existing approaches to software safety to reason about individual classes or components. It has been proposed in reference 2 that a way in which this could be achieved is through the use of *safety contracts* placed upon operations to constrain interactions between classes in the system, and therefore control functionality at a system level. In reference 2, it was discussed how the functional nature of many safety analysis techniques makes it difficult to elicit safety requirements for the contracts of individual classes. An approach based upon the analysis of interactions in the OO system was therefore proposed. This involves consideration of 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 a system hazard (ref. 2 and 3). The analysis involves timing analysis based on interaction diagrams, the use of fault trees, and analysis of state charts. The results of this analysis is then used to define the requirements that make up the safety contract.

Once the requirements for the system have been defined, the system must also be verified against these requirements to generate evidence that an implemented design meets these requirements. The evidence generated will form part of a safety case for the system. The safety case contains a safety argument for the system under consideration which demonstrates that the resulting system is safe to operate.Safety case argument can be represented effectively using the goal structuring (GSN) (ref.4). As for the safety analysis discussed above, it is critical that the safety argument developed can support change and reuse typical of an OO system such as the scenarios described earlier, otherwise the certification effort would negate the benefit achieved elsewhere. In the next section we look at how different approaches to constructing safety arguments affect the effort required due to the change and reuse scenarios described here. We then go on to look at the chosen solution in more detail and show how the safety contract based approach can generate the necessary evidence. Firstly, however, we introduce the concept of a safety case architecture.

Developing Safety Case Architectures for OO Systems

The Goal Structuring Notation:

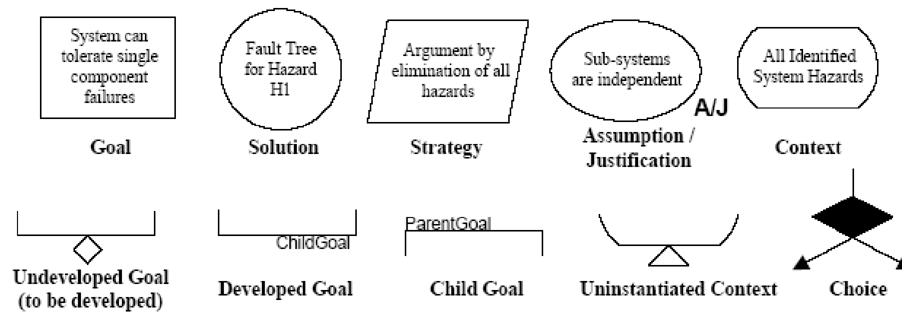

Figure 1 — GSN Symbols

The GSN symbols are shown in figure 1 and can be classified into the following categories:

- Argument Symbols - 'Goal', 'Strategy', and 'Solution' symbols
- Rationale Symbols - 'Context', 'Assumption', and 'Justification' symbols
- Linking Symbols - 'SolvedBy' and 'InContextOf' arrows

The argument symbols are used to show how the claims (**Goal** symbol) being made about the system are decomposed into sub-claims until a direct reference to available evidence (**Solution** symbol) can be made. Further clarification of the approach(es) taken can be provided by making explicit the **Strategy** taken to construct the argument. Argument symbols are linked using the **SolvedBy** arrow. The rationale symbols are used to provide an explanation for the stated argument. **Context** is used to link the argument symbols to information essential to their interpretation, e.g., definitions, system models, etc. **Assumption** symbols are used to link argument symbols to information necessary to understanding their validity. **Justification** symbols are used to link argument symbols to reasons why they are stated in particular manner. Rationale symbols are linked to the relevant argument symbols using the **InContextOf** arrow.

Safety Case Architectures:

In reference 5 Safety Case Architectures (SCAs) were introduced to facilitate the high level organisation of the safety case. In this paper we define a SCA in the following terms:

A Safety Case Architecture is the master plan for a particular safety case

A SCA is defined using the term master plan to indicate that it should provide safety case developers, maintainers, assessors, etc., with at least the following:

- The long term plan for the safety case
- Enough relevant information to ensure that any interested party can easily develop, locate, maintain, assess, etc., a particular part of the safety case

Although a SCA can be developed for non-modular safety cases, it is recommended that a SCA should be used to aid the development, and long term use, of a modular safety case. A modular safety case is where the safety arguments and evidence are arranged into modules. The modules should be developed so that they are as independent of other modules as possible. This approach is recommended as it should improve the potential changeability and reusability compared to a non-modular (or monolithic) safety case (ref. 6). The improved changeability arises from the independence of the argument modules, i.e, it should be possible to change the arguments in one module independently of other modules. The improved reusability arises from the independence of argument modules as it should allow better judgments to be made about how much one module relies on another, and hence, how reusable it is.

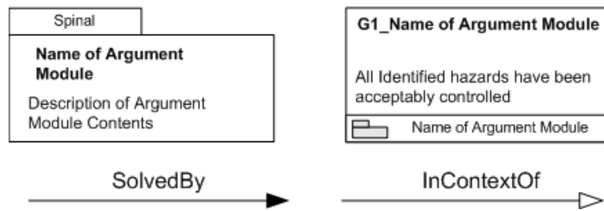

Figure 2 — Modular GSN Symbols

Part of the task of ensuring enough information is given is to provide the interested parties with a graph showing the SCA. This graph is used to indicate, amongst other things, the approach advocated by the SCA for the construction of a particular modular safety case.

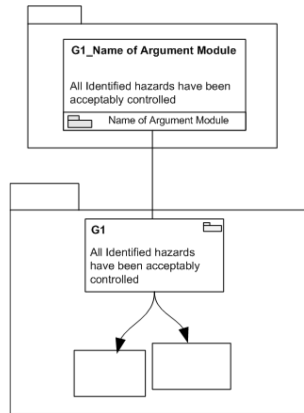

Figure 3 — An Example Away Goal

The graph can be constructed using the modular Goal Structuring Notation (GSN) symbols shown in figure 2. The **Argument Module** symbol (the folder like symbol) is used to build up a graph for the modular structure of the safety argument. Links using **SolvedBy** arrows, indicate that support is being offered by one argument module to another. Links using **InContextOf** arrows, indicate that one argument module is providing rationale for another argument module. **Away Goals** (the rectangle like symbol) to indicate direct links between argument modules as shown by figure 3

Other features of modular GSN include **argument module interfaces** and **argument module contracts**. The interface is used to make visible some of the contents of a argument module. The visible contents include all the rationale (reasons for why the argument is valid), intermodule dependencies (e.g., away goals), and the public claims or evidence contained in an argument module that offer (or require) support to (or from) other argument modules. Public claims and evidence are represented in GSN using argument symbols with the addition of a folder, usually in the top right corner of the symbol, such as G1 in figure 3. Interfaces are specified to provide other argument module developers with sufficient information to allow them use a particular argument module. The "sufficient information" is a judgement based on making a particular argument module as independent as possible, whilst providing enough information for the interface to be used. The contracts are used to capture links between the argument modules and the rationale upon which that link is based. Contracts are made using argument module interfaces, and are based on a number of things including: the consistency of rationale, the claims being made, and how strongly the offering claim supports the requiring claim. Contracts provide benefits such as making it easier to assess the impact, and potentially limit the effects, a change has on a particular modular safety case. These benefits are achieved by providing explicit breakpoints (i.e., the contracts) at which judgements can be made about whether a change presents sufficient challenge to the offering module such that its support for the requiring module is affected.

Comparing Alternative Architectures:

Here different modular safety argument structures are proposed for arguing the safety of an OO system. Each of these structures is qualitatively evaluated for its support of the change and reuse potential of an OO system. The first argument structure considered is the traditional, monolithic, approach to structuring the safety case. Using modular GSN this reflects an argument with one module that is responsible for the whole safety case and is shown in figure 4.

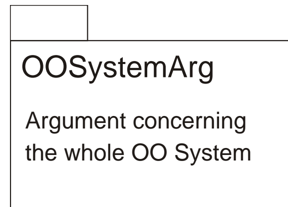

Figure 4 — Proposed Safety Case Architecture Number 1

Due to the potential web of argument dependencies commonly experienced by using this approach, any change to the design could impact the whole argument. It is therefore obvious that this traditional approach has poor support for change. In addition, it is also potentially very difficult to identify those bits of the argument that have been affected by any change as the impact could be spread throughout the argument structure. If a class were reused in another design, then with this argument structure it would be virtually impossible to reuse the corresponding bit of the argument in the argument for the new system. The argument is so highly inter-connected that separating out the argument for a particular class without loosing important elements of the argument for that class would be very difficult. Therefore, we contest that this traditional safety case approach is insufficient to retain the desirable features of OO systems.

The second argument structure considered is shown in figure 5. In this structure, the top-level argument module provides the scope for the overall safety case, and the child modules present independent safety arguments about the identified classes. Furthermore, each module is independently responsible for arguing about its interactions with all other classes.

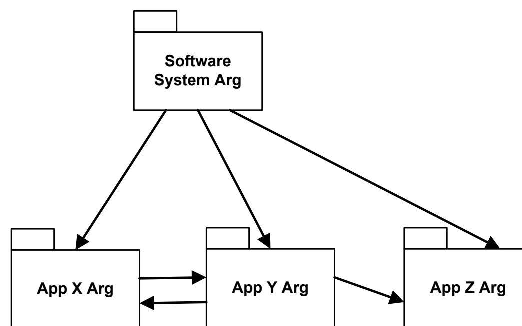

Figure 5 — Proposed Safety Case Architecture Number 2

Evaluating this argument structure for changeability and reusability reveals some interesting observations. This structure can be seen to have better support for change than the first structure, as the separate modules make it easier to identify the impact of a change on the argument structure. A change to the functionality of a particular class, for example should be more easily handled as there is separation in the argument. This means that the effects of the change can be contained within the module of the argument relating to that aspect without a knock-on effect on the rest of the argument modules. If we consider adding a new class to a system design however, this structure isn't so good. This is because the argument about interactions between classes in the system is contained within the class argument modules. Therefore, since the additional class affects interactions in the system, the argument in many modules could be affected. Furthermore, it was identified that the reuse potential of the argument modules is also limited by the dependencies made due to the arguments about the interactions. Since the module for a class reasons about interactions with other classes in the present system, it would require a great deal of reworking to use that module in a system where the interactions are with different classes.

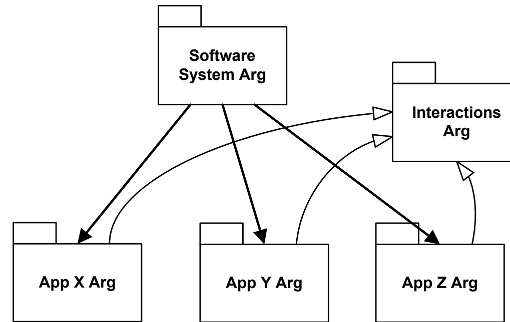

Figure 6 — Chosen Safety Case Architecture

The final argument structure shown in figure 6 represents the chosen architecture. The responsibilities are as identified for the previous structure, except here a safety case module has been added to make explicit the safety argument about the interactions between the classes. In this structure, an argument about **all** interactions is made in a separate module. The individual class argument modules now just make an argument that they will meet any constraints arising on them from these interactions. This means that the class arguments no longer have any responsibility to reason about their affect on other classes. This will make change much easier. Introducing a new class to the system for example will primarily impact just the interactions module, which is changed to handle the new interactions introduced. The structure also presents much better potential for reuse. Because a class argument no longer reasons about other classes in the system design (as it would if there were not a separate interactions argument module) it is much easier to reuse the argument module for that class in the argument for another system in which that class is also being used. For these reasons we propose that this argument structure is the most desirable for presenting a safety argument for an OO system.

Constructing the Modular Argument

In this section we will look in some detail at the arguments that are presented in each argument module of the chosen architecture and how the modules are linked via module interfaces using contracts such that an overall system argument is formed.

Software System Level Argument:

Firstly we will look at the Software System Level Argument Module. This is the top level argument which argues that the software is acceptably safe and can be seen in figure 7.

The way in which the safety of the software is demonstrated will be through a hazard control approach. The reason this approach is taken is that the proposed analysis of the OO system mentioned earlier takes a top-down approach starting from the system level hazards and identifying how the software may contribute to these hazards. The safety argument therefore reflects this approach. The argument then shows that the software contribution to each system level hazard is acceptable. This is done by arguing that all the Hazardous Software Failure Modes (HSFM)(that is ways in which the software might fail to bring about the hazards of interest) have been identified correctly and will not occur in or are mitigated by the software. It is worth noting that the HSFM are high-level failures at the software system level so identifying these should not require knowledge of under-lying software system design. The process for identifying HSFMs will include Functional Failure Analysis of system level functions.

In order to show that the HSFMs do not occur, the strategy adopted is to argue separately that the interactions are safe and that the individual classes themselves will not behave in such a way that they will contribute to the HSFMs. This strategy fits in with the decision taken earlier about the preferred safety argument module structure (figure 6). It is also necessary to argue that there are no unintended interactions between classes in the system. The undeveloped goals G5_Sys and G6_Sys are public goals which need to be addressed by other modules in the argument (in this case the interactions argument and the argument for the class). This information is captured in a safety case module interface for the software system level argument. Safety case module contracts will be developed to capture the way in which

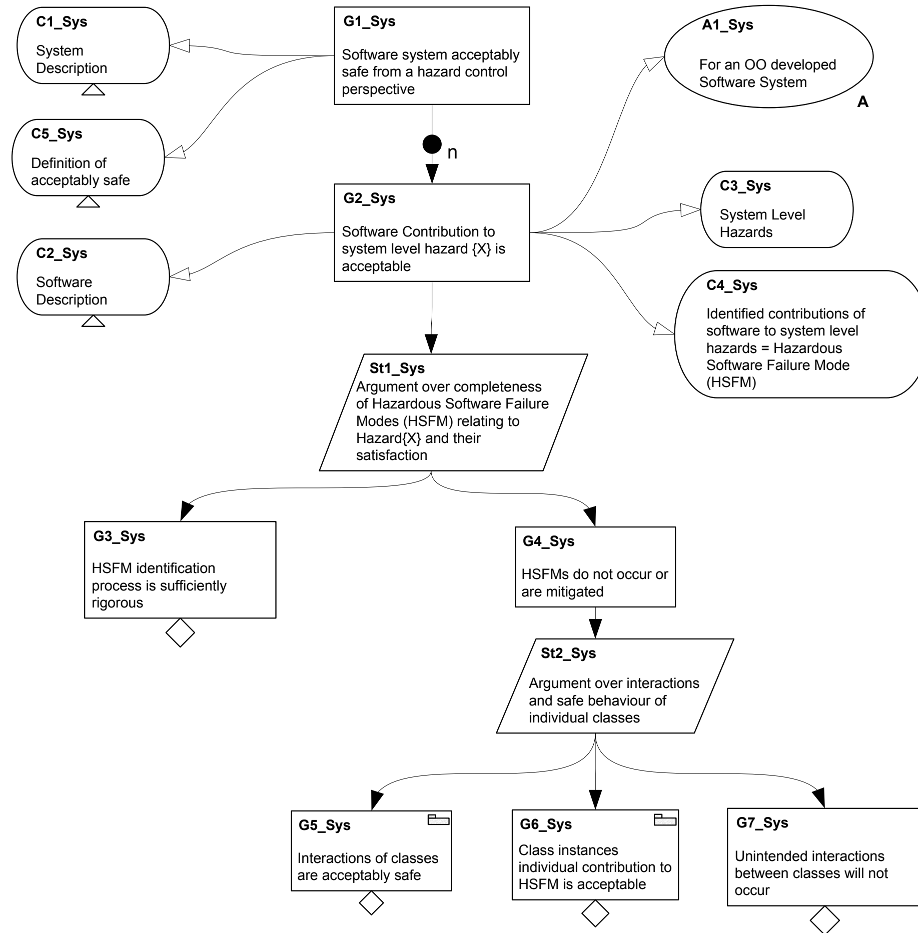

Figure 7 — Software System Level Argument

the public goals for this module are satisfied by the other modules.

Interaction Argument:

Now we look at the Interactions Argument Module which can be seen in figure 8.

This argument module argues that the interactions that occur between classes in the system are acceptably safe. This is done by arguing firstly that the contracts have been correctly identified, and then that the requirements arising from those contracts have been satisfied. We show that the contracts are identified correctly by arguing that the relevant interactions for each of the identified HSFMs have been identified. Then it is shown that for each of these relevant interactions, the operations called have contractual constraints identified necessary to ensure that the HSFM will not occur. This is argued by appealing to various forms of evidence which are generated from the analysis process. This includes evidence from fault tree based analysis, timing analysis of interaction diagrams and analysis of statecharts. Full details on this analysis is contained in reference 3.

It is then argued that all the classes involved in interactions for which safety contracts have been specified can meet those requirements. This argument is made through goal G3_Int, which is a public goal which will be addressed by the individual class argument modules. By doing this, it is ensured that the argument about the behaviour of the classes is separated from the argument made about the interactions. Again, this strategy fits in with the structure of the preferred SCA in figure 6. As with the software system argument module, this dependency is captured in the module interface and contract.

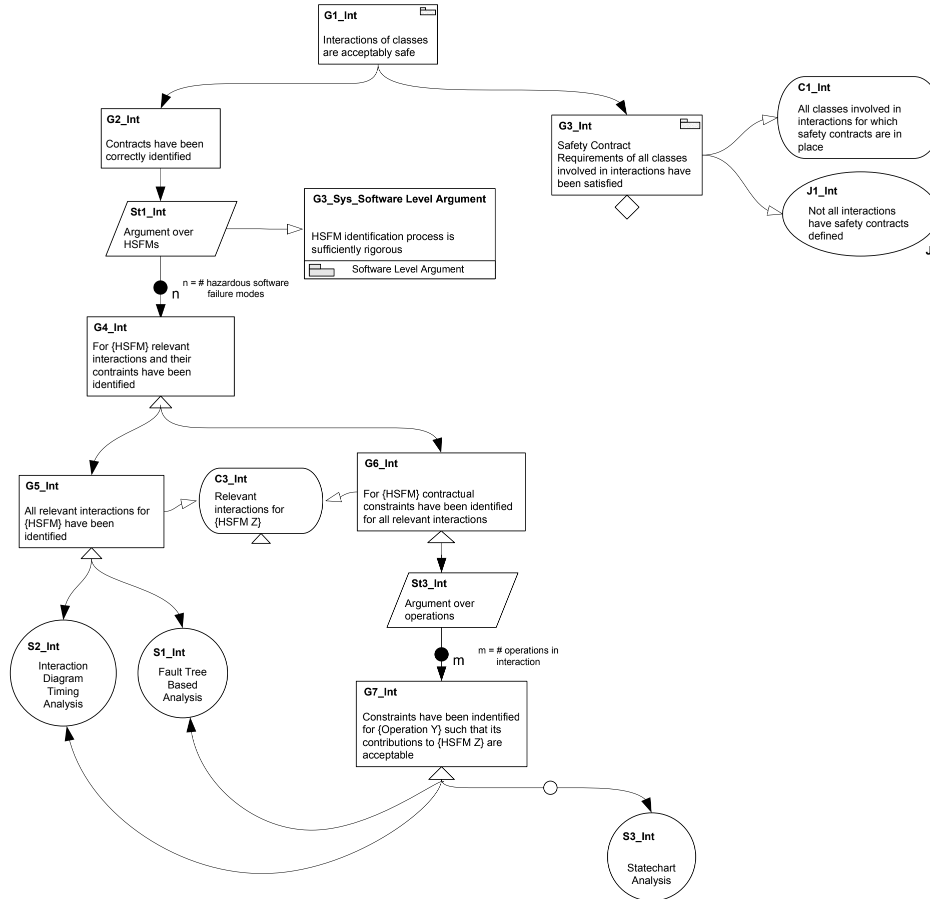

Figure 8 — Interactions Argument

Class X Argument:

An argument module is developed for each class in the system design. The argument for each class is as shown in figure 9.

The purpose of the class argument is to show that the class will not contribute to any of the relevant HSFMs. This is done by arguing that the requirements on each instance of that class (object) arising from its contractual constraints have been satisfied. This argument satisfies goal G3_Int that required support in the interactions argument. The relationship between these goals in the two argument modules is recorded as a safety case module contract.

The goal G2_Class argues that the safety requirements from the contracts have been satisfied. This involves identifying all the contractual requirements from the operations in which that object is involved. An object may be involved in an operation as the client of the interaction (that is the caller), or as the supplier (that is the callee). The client must meet the preconditions of the interaction, whilst the supplier has responsibility for the postcondition. It is then argued that the class will satisfy all these requirements derived from the contractual constraints. This could be done by appealing to evidence such as testing or static analysis.

It is also necessary to argue that the class itself will not behave in such a manner as to contribute to the HSFM. This is to cover behaviour which classes may exhibit other than as a result of interactions with other classes. This will be, mainly, such things as subtle value failures which could be present in an implemented system. Goal G3_Class provides this argument. Again this goal addresses a goal requiring support in another module (software system level argument) and a contract will be constructed to capture this relationship. This goal also requires further development to show

how such contributions are identified.

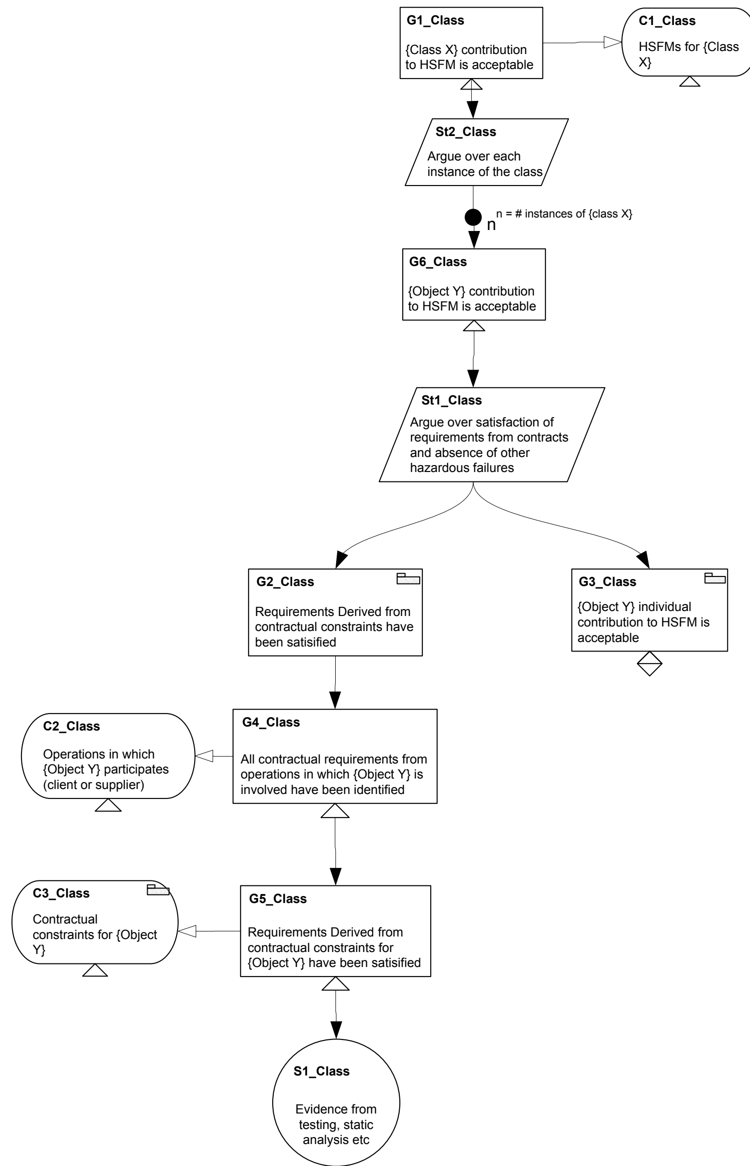

Figure 9 — Class X Argument

Conclusion:

In this paper we have identified the need for an alternative to a monolithic safety case argument structure to support the use of an OO approach to safety critical systems development. We have proposed a modular safety argument approach as it offers the flexibility required to support many of the features that characterise OO systems. Some alternative modular argument structures were proposed, and each was assessed for its ability to support some defined change and reuse scenarios. Based on this a preferred architecture was chosen. We have then shown how the arguments presented in each module may be developed.

References

1. B. P. Douglass. Doing Hard Time - Developing Real-Time Systems with UML, Objects, Frameworks, And Patterns.

Addison-Wesley, 1999.

2. R. Hawkins, J. McDermid, and I. Bate. Developing Safety Contracts for OO Systems. In 21st International System Safety Conference, 2003.
3. R. Hawkins, I. Toyn, and I. Bate. An Approach to Designing Safety Critical Systems Using the Unified Modelling Language. In Critical Systems Development with UML - Proceedings of the UML'03 Workshop, 2003.
4. T. Kelly. Arguing Safety - A Systematic Approach to Managing Safety Cases. PhD thesis, Dept. of Computer Science, University of York, 1998.
5. T. P. Kelly. Concepts and Principles of Compositional Safety Case Construction. Technical Report COSMA/2001/1/1, The University of York, 2001.
6. J. Rushby. Modular Certification. Technical Report NASA/CR-2002-212130, Langley Research Center, 2002.

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.

S.A. Bates, MEng, Research Associate, Department of Computer Science, University of York, Heslington, York, YO10 5DD, UK, Tel: 0044 (0) 1904 433385, Fax: 0044 (0) 1904 432708, Email: simon.bates@cs.york.ac.uk

Simon Bates has been a Research Associate in the BAE SYSTEMS funded Dependable Computing Systems Centre (DCSC) at the University of York since October 2002. He graduated from the University of Manchester in 2002, where he attained a MEng (Hons) in Electronic Systems Engineering. Since taking up his role with the DCSC he has developed the following research interests: Safety Cases, Safety Case Architectures, Modular and Incremental Certification of Safety Related Systems, and Software Architectures.

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.