

Safety Assurance Contracts for Integrated Modular Avionics

Philippa Conmy, Mark Nicholson, John McDermid

Department of Computer Science

University of York

Heslington

YO10, 5DD

United Kingdom

{philippa.conmy, mark.nicholson, john.mcdermid}@cs.york.ac.uk

Abstract

This paper describes a method for performing safety analysis on an Integrated Modular Avionics system in a manner which supports the incremental development and change of system components. This is achieved by analysing each component in the context of the overall system design and then finding derived safety requirements. Each IMA component (hardware, software or both) is then examined to determine how these safety requirements are met, and a contract is formed which captures the rely/guarantee conditions between that component and any component which relies on it. This contract captures the behaviour which must be preserved by a component following incremental change.

Keywords: Integrated Modular Avionics, Safety, and Modularisation.

1 Introduction

The aviation industry, both in the military and civil domains, is moving towards widespread use of distributed computing networks. This is commonly known as Integrated Modular Avionics (IMA). Within IMA systems, avionics applications are supported by generic resource management software (such as Operating Systems) which control access to computing resources such as processor time and memory by applications. The supporting software also enforces strict partitioning of resources to prevent interference between applications, e.g. to prevent a misbehaving low integrity application from interfering with the operation of a high integrity, safety critical application.

The components of an IMA system are often purchased and developed incrementally. For example, the supporting Operating System (OS) may be developed first in order to provide a realistic environment for the development of applications later. To ensure interoperability, IMA systems use common interface standards between software and hardware components. The use of standard interfaces means that components can be altered or inserted into an existing system without

requiring additional alteration of the original components. Whilst these interfaces capture syntactic requirements (such as range and types of variables), behavioural details (such as response to failure) are not always as well defined. In addition the internal behaviour of many components, such as operating systems which manipulate other components without requiring a direct software call or communication, is not well defined.

It is important to understand how components can affect one another if system safety is to be assured. In addition, a modular approach to capturing safety requirements is needed to support the incremental development and incremental change of components. Existing techniques are not well suited to capturing safety dependencies in a format which supports incremental change. This paper describes a modular technique, looking at how the supporting software of an IMA system meets one of its safety requirements: memory protection. This technique captures behaviour of the software when meeting its safety requirements, and examines how this may affect the safe operation of another component. Each of the other components designers can then either assess whether this is acceptable, or negotiate until it is acceptable - forming a "contract". This approach supports both the incremental development approach and incremental change by capturing the behaviour that must be preserved when a component is altered. If the behaviour is still assured (and no new behaviour introduced), system safety is retained.

The next section of this paper describes the architecture of a proposed industrial IMA supported navigational display system. This section also describes the problems encountered when analysing components independently of one another. The third section describes how system safety requirements may be derived, and then describes in detail how to examine each component to assess how they meet their requirements. The fourth section describes some software designed to be used in an IMA module. The fifth section demonstrates how a contract may be derived for this software, particularly for memory partitioning. The following section discusses how to resolve outstanding issues in the contract. Finally, discussion, conclusions and acknowledgements are presented.

2 System Description

There are two main designs used for Integrated Modular Avionics. In the civil domain the ARINC 653 standard is used (ARINC 1997). In the military domain the ASAAC

standard is used (Multedo et al. 1999). The system examined for this paper was a proposed navigational display system with a single application to be embedded on a computing platform based on the military ASAAC style IMA. The IMA architecture is shown in Figure 1. Each box within the diagram represents an individual IMA component.

The top layer of application software is divided into a number of partitions of single integrity. This software is aircraft dependent and hardware independent. This means that it is written for a particular aircraft, but contains no code which specific to a particular computer processor. The next layer undertakes all requests for processing or communications. This is the Operating System (OS) Layer (OSL) which contains three main sections. The OS, which manages tasks such as communications requests, the General System Manager (GSM) which provides data such as which communications are permitted, and the blueprint (BP) manager which controls the layout of partitions. The OSL is both aircraft and hardware independent.

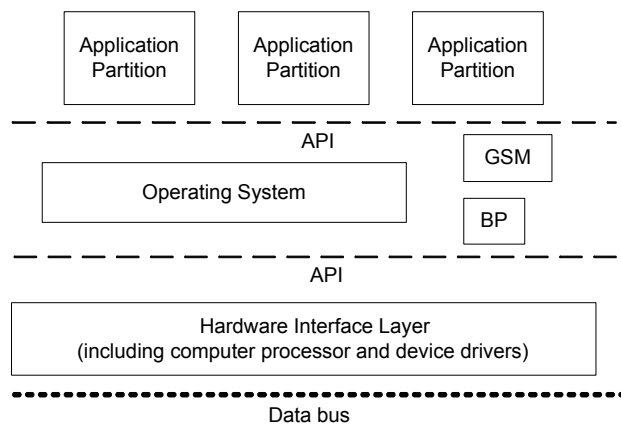


Figure 1 IMA Module Architecture

The bottom layer is the Hardware Interface Layer (HIL) which contains low level software which interfaces with the computer processor. This manages memory and access to processing facilities. This layer is hardware dependent and aircraft independent. Function calls are made between each layer using a standard Application Programming Interface (API). This three-layered approach allows incremental change in one component to be supported with minimal impact on other components. For example, the use of the HIL means that the computer processor can be upgraded without affecting the software of the other components (although performance details such as execution times may change). This helps combat the ongoing problem of hardware obsolescence within the aviation industry. Another example is the alteration of an application partition to fix a software bug. As long as the applications scheduling requirements can be met then no alteration is required to the other components in the system.

The navigational system design consists of a number of input sensors, a number of application partitions (which calculate display data such as attitude and altitude), a small network and a display device. The supporting IMA platform is being purchased prior to the full development

of the navigation application. This means the application can be developed using its execution environment. In order to ensure that system safety can be achieved once the components are integrated, however, some safety analysis needs to be performed on the HIL by system designers, and the resulting safety requirements included in the HIL requirements documentation.

Analysing the components independently of one another is not enough to assure system safety due to the highly integrated nature of IMA. A failure which manifests itself in one component may have actually been caused by another (for example the HIL may detect an application attempting to divide by zero or an application may attempt to use a communications channel which it does not have permission to use). In addition, it may not be appropriate for the component which detects a failure to take action to resolve it.

As stated previously, the HIL is designed to be aircraft independent. This means it can be used in many different IMA systems on the same or different aircraft. There are a number of software products already available which can be used to provide the HIL functionality. In each case the softwares suitability needs to be assessed for a particular context.

The HIL manipulates all the applications by copying them in and out of memory for scheduling purposes, and by managing data interchange during execution. It does not, however, directly make a call to any application, therefore, any behavioural side effects which may affect an application, such as responses to misbehaving applications, need to be captured. The next section summarises a process which can be used to capture and summarise this information.

3 Analysis Process

The proposed IMA safety process is divided into two main phases. In the first phase, system analysis provides the high level safety requirements for each of the components. In the second phase, each component is examined to see how these requirements are met. The first phase has been applied to the design of the navigational system, identifying a number of key requirements for the HIL. The second phase has been applied to a version of the L4 microkernel, adapted specifically for use as a HIL for IMA. By applying the safety requirements from the system design, the suitability of this software for purpose can be assessed.

3.1 Identification of dependencies

The first phase in the safety process involves the identification of safety dependencies between components. This phase is divided into three parts. First, some high-level system analysis is undertaken, including identification of system level hazards, production of an architectural structure diagram, and summarising the role played by each component in the structure. Second, each component is analysed individually to see how failures within that component could impact system safety, and to derive appropriate mitigation strategies (known as derived safety requirements (DSRs)).

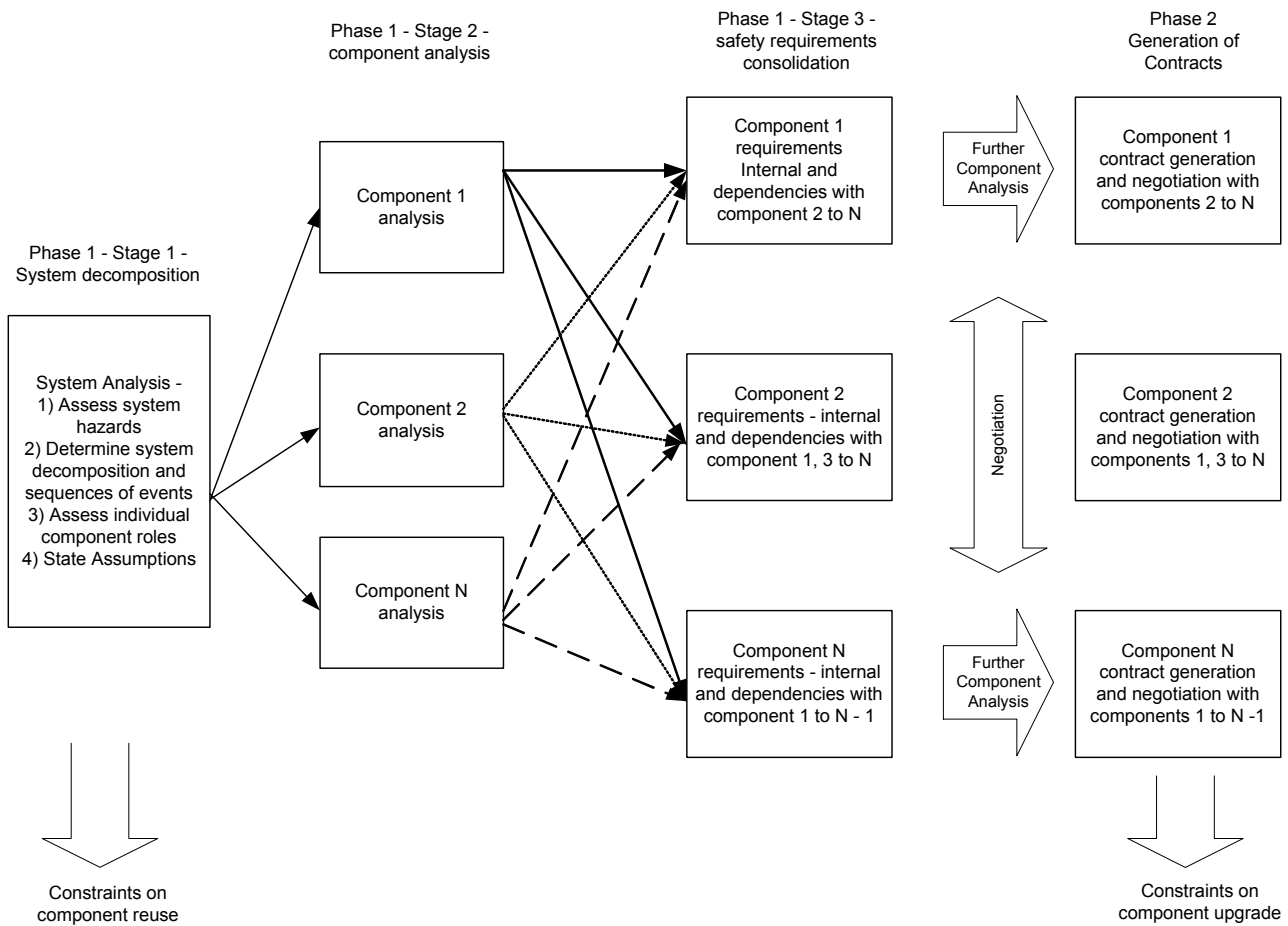


Figure 2 Summary of Overall Safety Process

Performing this component analysis within the context of system level analysis can mean that components require re-evaluation when being re-used within a different scenario. The need to provide assurance of the safety of the current system generally outweighs the need to consider future deployment of components.

Finally, a consolidation process is undertaken to ensure the DSRs are consistent. Three types of requirement are found:

- Requirements from the components analysis which address internal failures
- Requirements on the component which address external failures (in other components)
- Requirements on the component which are the results of another components analysis.

Additionally, there may be "if - then" requirements where the results depend on behaviour of another component. Where one component mitigates against a failure in another component, or protects another component from a failure, a safety contract is formed.

An overview of this phase is on the left hand side in Figure 2. (A full description can be found in (Conmy et al. August 2003)).

3.2 Development of the contract

The second phase (shown on the right in Figure 2) involves the development of a contract for each of the consolidated DSRs. There are many computing fields which use the concept of a contract to capture the relationship between two entities, e.g. Object Oriented Programming (Meyer 1992) and Formal Methods (Jones 1983). A contract is a set of rules constraining the interactions between a supplier of services and all its clients. The rules state the benefits and obligations of each party when participating in a relationship. These rules are divided into:

- pre-conditions (which every client using the service must adhere before the supplier can function),
- post-conditions (the state of the system after execution),
- rely conditions (what assumptions are made about the rest of the system), and
- guarantee conditions (what the supplier provides during its execution).

The idea of a safety assurance contract is to identify the constraints on a key safety guarantee. This involves first

identifying the supplier component which meets that guarantee, then examining how the supplier meets that guarantee in order to find the conditions under which it can be upheld. This differs from a software contract in that the guarantee is often met by more than one function within a component, and may rely on software normally hidden from the client. For example, the context switching software of an OS copies application software in and out of memory at the beginning and end of a scheduling slot. This software is not normally visible to a client application; however, the application must rely on its correct function in order to meet its safety and reliability requirements.

The contract provides the framework within which the client and/or supplier can be upgraded. As long as the guarantee conditions are still met by the altered component then system safety should still be assured. This helps support the modular approach of IMA.

Once the first phase of the safety process has been completed, the contract needs to be developed to examine how a component meets its DSRs. It should be noted that simply developing the components individually to meet the DSRs will not guarantee safe interoperability; there may be side effects and constraints generated by the way a component deals with the failure conditions identified (pre-conditions). For example, if an OS must halt an application following a processing hardware failure the application designer must make adjustments in order to handle the situation e.g. using backup lanes.

The contract can be developed at different levels of abstraction. For the purposes of this paper, the following four levels are considered appropriate:

1. The high level requirements from the first phase
2. Architectural constraints covering, for example, any hardware choices made which will impact on the structural design of other component
3. Behavioural constraints, looking at the functionality of component
4. Quantifiable details such as syntactic requirements, performance or reliability requirements

The constraints identified at each level form a set of guarantees in that the client, supplier or both must agree to meet them.

At each stage the client component designer may need to agree to any identified constraints before further system development. To this end each constraint should be assessed as to whether it is deemed acceptable by both client and supplier designers. The suggested categories for this process are: proposed, obligation, accepted, accepted obligation, and challenged. The proposed category indicates that this constraint is flexible and the client or supplier designers may be able alter it. The accepted category indicates that a proposed constraint has been accepted. The obligation category indicates that the constraint cannot be altered and that there is a requirement on the designer of the affected component(s) to design accordingly. The accepted obligation category

indicates that an obligation has been assessed as acceptable. The challenged category can be used after an incremental change, and indicates that the listed constraint may have been broken. Note that in the case where an obligation is not acceptable the supplier may require an alternative design strategy. The use of these categories is demonstrated in sections 5 and 6.

3.2.1 Contract Development Process

To derive a contract the following process should be undertaken for each DSR which forms a contract between two or more components. The analysis process is described for each of the four levels of abstraction.

1) List all client components which rely on the supplier in order to determine the range of impact of development.

2) Perform Architectural Level Analysis as follows:

- Determine architectural decisions made which impact on the high level requirement
- Summarise into a set of "guarantee" conditions for the supplier.
- Assess whether these guarantees impact on clients. If so, express impact as "rely" conditions. Mark flexible conditions as "proposed", inflexible conditions as "obligation", and conditions with no impact on clients as "accepted".
- Assess acceptability of "rely" conditions with client designer. If flexible conditions are acceptable then mark as "accepted", if not then some negotiation with supplier is required. If obligations are acceptable then mark as "accepted obligation", otherwise assess possible client changes which could be made to ensure compatibility or consider whether client can be used in the system.

3) Perform Behavioural Level Analysis as follows:

- Analyse the behaviour of the supplier component which can impact on the high level requirement including failure behaviours of the component.
- Summarise into a set of "guarantee" conditions for the supplier.
- Assess whether these guarantees impact on clients. If so, express impact as "rely" conditions. Mark flexible conditions as "proposed", inflexible conditions as "obligation", and conditions with no impact on clients as "accepted".
- Assess acceptability of "rely" conditions with client designers. If flexible conditions are acceptable then mark as "accepted", if not then some negotiation with supplier is required. If obligations are acceptable then mark as "accepted obligation", otherwise assess possible client changes which could be made to ensure

compatibility or consider whether client can be used in the system.

4) Perform Quantification Level Analysis as follows:

- Analyse the syntactic requirements of the supplier component which impact on the high level requirement. These are unlikely to be flexible, and it will merely be a case of ensuring the client component meets the specified function interface requirements.
- Analyse the performance requirements of the client component which impact on the high level requirements. This information is largely client driven, as the supplier is providing something for the client. Using ranges rather than absolute values for the requirements gives flexibility, not only during design and implementation, but also for incremental change of components.
- Analyse the reliability requirements of the client component which impact on the first level requirement. This information is largely client driven, as it is a failure mode being addressed for the client.

Note that the reliability requirements are difficult to assess particularly if there are multiple clients relying on the same feature. In addition, a purely software component will have no specific reliability values associated with it.

A suggested technique for deriving performance data in the quantifiable level is reservation based timing analysis (RBA) (Grigg and Audsley 1999) in which timing budgets are allocated for each component in a system. As it is developed these budgets are gradually made concrete and as long as each component is within its budget the overall timing constraints can be met.

The results of the analysis should be summarised within a table (such as that given in Table 1). Data at each level may need to be examined during any stage of development for a particularly important requirement. However, if the data is captured in this structured manner, it assists incremental change. After a change, the contract is "rolled back" a level at a time to assess the impact of the change. For example, if the computer processor is altered, then only the quantification level may need to be examined.

4 Introduction to the Hardware Interface Layer

This section first summarises the types of DSRs which were found for the HIL, then describes the proposed HIL software.

4.1 Safety Requirements

The first phase of the safety process examined the navigation system design in order to provide the context for analysis of the HIL design. The suppliers and designers of the IMA system met together to perform the first stage of this analysis to avoid misunderstandings

about the design. A simplified architectural system design diagram was constructed, overlaid with arrows indicating data flow. The suppliers specified the function of each component in the system. The HIL was then analysed.

The HIL directly controls the scheduling of both the applications and the OSL. API calls are only made to the HIL by the OSL; the applications are not allowed to make direct calls to the HIL. The HIL is responsible for copying data between application partitions and also for managing the applications access to the processor and memory. The HIL is also responsible for passing data on to the navigational display following calculation by the application. This means that data corruption by the HIL can directly lead to incorrect data displayed to the pilot, therefore one of the derived safety requirements for the HIL concerns preventing corruption of data sent to the network. This DSR demonstrates the need to consider the operating environment of the HIL in order to assess its suitability.

A number of DSRs for the HIL were found, and a more extensive description of the HIL analysis can be found in (Conmy, Nicholson et al. August 2003). One of the key DSRs is that the HIL must prevent memory writes outside of an application's designated area. This helps prevent a number of failures including context switches to the wrong application and corruption of data to be sent to the pilot display. This requirement is also commonly known as "memory partitioning". This feature is now examined in order to develop the contract.

4.2 HIL Description

The HIL analysed was a version of the L4 microkernel (Heiser 2001) developed specifically for the IMA environment (Bennett and Audsley 2001). The L4 microkernel specification describes a small set of functions for Inter-Process Communication (IPC) and scheduling. This core functionality supports access to the processing resources and hides the underlying hardware. It does not limit access to these resources but instead provides support for the OSL to implement its own security and management policies. This makes it ideal for use in an IMA system where the OSL and applications may be differently configured for each IMA module.

The kernel examined provides strict partitioning of memory spaces for the applications and OSL. The kernel has been written for the PowerPC 603e using a combination of the C programming language and assembler code. Whilst this means the code is not portable, it is necessary in order to achieve the performance required for a real-time system. The version analysed for this paper was a partial implementation of the full L4 function set, which meant that there was some room for negotiability when deriving the contract. The full source code was available for inspection during the second phase of analysis.

4.2.1 Memory partitioning

The L4 microkernel examined uses the Memory Management Unit (MMU) on the Power Pc processor in order to enforce memory partitioning. Computer memory

is divided into sections known as pages or segments which contain either data or instructions (for execution). These are swapped in and out of secondary storage to physical memory as and when they are required during an application's execution. Pages have a fixed size whereas segments can have a variable size, the advantage of the latter being that more efficient use of memory can be made. However, pages are simpler to analyse. The kernel examined used pages only. Software accesses a specific memory location using a page identifier and an offset from this identifier. This is known as an address (see Figure 3).

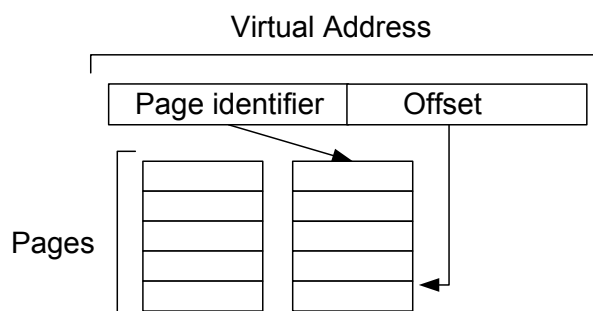


Figure 3 Virtual Address Mapping to Physical Pages

A MMUs function is to translate a virtual memory address into a physical memory address. This allows a computer application to use more memory than physically exists by allocating physical memory only when it is required, and de-allocating memory as an application no longer requires it (i.e. removing instructions/data that are not being used at present). Various memory attributes and settings can be utilised to limit access to memory areas, and the MMU (sometimes supported by software) will raise an exception when an attempt is made to access data in an area which would violate the access permissions. This functionality can be used to provide a memory partitioning system.

The MMU keeps a list of recently accessed (virtual) pages in a Translation Lookaside Buffer (TLB). A full list of pages for an application is kept in a page table, however this is often large and awkward to search, hence the use of the TLB to reduce average case look up time. When application code uses a virtual address to access memory the MMU looks for a match in the TLB. If there is none, an exception is generated. The HIL will intercept this in order to examine the page table. If the page is found in the table then it is inserted into the TLB, if not a page fault exception is raised. An exception handler, part of the HIL, will process this event.

One form of memory protection uses multiple levels of security, commonly supervisor and user levels, to limit access. The HIL runs in the supervisor mode and can access all memory areas. The applications run in user mode and can only access a restricted set of memory areas. Note that this means the HIL must be trusted not to alter the application's memory space.

Usually, each applications virtual address space starts at 0. This means that different applications use the same virtual addresses to access memory (it is the mapping to physical memory that changes). If, after a context switch

(where one application becomes active and another inactive) an entry is left in the page table from the last application, the new application could potentially access the old applications data. Therefore, a mechanism is required to ensure that the different user applications do not access each other's memory space. This is often achieved using an Address Space Identifier (ASID). This ASID is included with the virtual address. When an attempt is made to access an address, the ASID of the requesting application is checked against the ASID of the address in the memory map and an exception raised if they don't match. Alternatively the entire TLB and page table can be emptied to ensure that no entries are left before switching between applications. This has an impact on overall performance, but ensures that no entries remain in the page table thus allowing accidental access to another partitions data.

The memory partitioning functionality in this kernel uses normal virtual addressing for applications with soft or no real time requirements. In addition, the kernel makes special provision for applications with hard real time requirements by taking advantage of the PowerPCs Block Address Translation (BAT) registers. These allow areas of memory to be ring-fenced, with direct translation of addresses via the BAT registers rather than the TLB. This means that the conversion time from virtual address to physical address is bounded and predictable, with the application never suffering a page miss. This is useful when calculating the Worst Case Execution Time (WCET) of an application. However, the ring fenced areas have a limited size of 256K meaning the application running in these areas must have a memory footprint of this size or less (note that is part of the quantifiable level of the contract).

5 Deriving the Contract for the HIL

This section examines how the bottom three levels of the contract can be derived. The first level of the contract as described in section 4.1 states that the HIL should prevent corruption of an applications memory space. The HIL is the supplier of this service and the applications and OSL are clients. Note that this guarantee looks only at how partitioning may be broken, not at other features of memory provision such as speed of access; this would be a different contract.

5.1 Architectural Level

For the architectural level of the contract, the decision about the type of mechanism to be used to protect the application's memory space is examined. In this case the MMU from the PowerPC is to be used meaning that each application will need to be constructed from one or more single integrity partitions. The application provider needs to design and examine their system to ensure that this constraint will be acceptable. As shown earlier, the IMA concept demands that applications be designed this way, therefore this method can be marked as accepted by both supplier and client. This level of the contract is shown in Table 1.

Table 1 Architectural Level Contract

Architectural level constraints				
Potential Failure	HIL Guarantee (supplier)	P/A	Application(s)/OSL rely (client)	P/A
Memory space alteration by external source	Will provide mechanism which supports single integrity level partitions	A	Demonstrate that application is acceptably safe divided into multiple single integrity partitions	A

5.2 Behavioural Level

This section presents the results of the behavioural level analysis, and puts it into the proposed contract format. The summarised contract is shown in Table 2.

In order to develop the behavioural level of the contract the kernel code (C, and assembler) has been examined to see how the code upholds this guarantee, and also to see whether the guarantee could be broken. One of the first considerations is the MMU protection mechanisms used. Firstly, although unique ASIDs are used for both applications and the HIL there is a potential loophole which could break the partitioning. The HIL does not check the ASID unless there is a page miss. In other words, if a request is made for an address, and there is a match in the TLB then the memory is accessed directly. This means that after a context switch from one application to a new application, there may potentially be mappings to the previous applications address space left in the TLB. The new application could then access this memory space accidentally. To ensure this doesn't occur, all parts of the HIL code which perform a context switch has been examined to check that the TLB is emptied (known as flushing).

An additional concern might be that an application could alter its ASID somehow. In order to prevent this, the ASID is only ever stored in a supervisor controlled area of memory which the applications and OSL have no access to.

These mechanisms prevent any memory violation for all applications using normal memory space and have no side effects on the applications. They are therefore marked as "accepted" in the contract. It should be noted that the HIL does not attempt to detect whether these applications have attempted to violate partitions, rather it prevents any violation from occurring. This means no data can be gathered on whether the applications are misbehaving.

The next consideration is the special ring fenced areas. As the code in these areas should be fully mapped prior to execution, any page miss can be viewed as an attempted partition violation. The version of the kernel analysed for this paper at present fails silent (within an infinite loop) if this situation occurs. If this action is chosen for the final version of the kernel then each application supplier must provide evidence that fail silent is an acceptable failure condition. Clearly, a better strategy is that the HIL only

closes down the offending partition. In either case the action taken places a safety constraint on one or more client applications. This information is captured in the contract and marked as "proposed" in order to show that the alternatives should be examined further.

Table 2 Behavioural Level Contract

Behavioural level constraints				
Potential Failure	HIL Guarantee (supplier)	P/A	Application(s)/OSL rely (client)	P/A
Applications with identical virtual address space numbering access each others memory space	Unique identifier is used for each application. Kernel performs fail silent in the event of a memory violation in one of the ring-fenced areas	P	Ensure the ring fenced applications do not violate their boundaries using static analysis <u>OR</u> All applications need to ensure fail silent is acceptably safe.	P
Applications change their identifier	Address space identifiers protected using supervisor privilege	A		
Applications gain supervisor privilege	HIL ensures application ID in place when application is executing	A		
The HIL alters the applications memory space	HIL uses unique ID	A		
Application gains direct access to another address space via IPC	Either – implementation does not allow memory sharing <u>OR</u> HIL supplier provides restriction rules for set of IPC functions	P	Ensure IPC instructions adhered to.	P
Memory areas are initialised incorrectly	HIL performs check on startup	A		
Changes made to memory permissions during execution do allow applications to access another's memory space	No changes made.	A		
The HIL is altered by applications	HIL is protected using supervisor privilege.	A		

The other potential partitioning problem is that, according to the L4 standard (Heiser 2001), some IPC calls allow memory areas to be shared between applications. This is a more efficient way to share data between applications rather than via a third party. This approach is not necessarily suitable for an IMA environment, however.

Firstly, it potentially allows the partitioning to be broken. Secondly, if two partitions directly share memory with one another they must be located on the same processor, placing a restriction on the configurations of the system. Clearly, there may be a situation where the only way to achieve the performance required to meet a hard deadline is by sharing memory. It is therefore proposed that the implementation either does not allow memory sharing, or that if it does a set of clear restrictions on how the IPC may be used is provided to the application designers so that they can avoid any potential problems. Again, this information is marked as "proposed" within the contract.

The last two paragraphs indicate the importance of this process. As part of the HIL implementation there are side effects which can impact the applications. The application designers need to be aware of these issues during their own development process in order to design their software within the constraints.

5.3 Quantifiable Level

Finally, the quantifiable aspects of the contract are examined. At this stage only the type of evidence required and how it could be generated is discussed as there was not enough system information available to make a more detailed assessment. This level of the contract is shown in Table 3.

Quantifiable level constraints				
Quantifiable feature	HIL Guarantee (supplier)	P/A	Application(s) /OSL rely (client)	P/A
Application require real time memory spaces	HIL provides real-time spaces of size 256 K	O	Applications memory footprint must be no greater than 256K.	P
Number of partition failures per flight hour	MMU fails with probability of 10^{-9}	P	Applications have not greater reliability requirement (in combination) than 10^{-9}	P

Table 3 Quantifiable Level Contract

The memory partitioning mechanism does not require any specific API calls to be made; therefore no syntactic requirements are needed.

In terms of performance, the relevant information is that the real time address space areas have a size limitation of 256K. This limitation is a property of the processor and, thus, cannot be altered. It is marked as O for "obligation" in the contract.

In order to determine the reliability requirements, the supplier would need to know the full range of applications executing on the IMA system in order to determine the worst case failure. This means examining all the components in the system which it relies on. Unfortunately this data is was not available at the time of writing the paper, therefore a reliability requirement of

10^{-9} per flight hour has been proposed. This requirement can be examined later on in the development process.

6 Consolidating Applications and OSL to the Contract

This section examines the open issues from section 5, by comparing applications and OSL designs against the contract, looking those rows marked as "P" for proposed and "O" for obligation.

There are no open issues at the architectural level of the contract.

The behavioural level of the contract has two rows marked as proposed. Firstly, the response to detection of a page miss in one of the real time address spaces is to cause the whole processing module to fail silent. The navigation system has both a backup lane for the main calculation application and an independent backup system. This row can, therefore, be marked as "accepted".

The other open issue at this level is whether IPC allows memory to be shared. According to the IMA philosophy only the OSL is allowed to make direct calls to the HIL. As the OSL in this design does not require IPC the option of not implementing memory sharing in the final version can be taken, and the row marked as "accepted" for both client and supplier.¹

At the quantifiable level two issues remain. As previously discussed the issue of reliability cannot currently be resolved. This leaves the constraint that the applications in the real time address spaces are limited to a 256K size. As the final design, and partition layout, of the application is not yet finalised, the application can be divided up such that this limitation is acceptable. This row can therefore be marked as "AO" for "accepted obligation".

7 Discussion

This section discusses a number of issues arising from the analysis process.

One of the advantages of deriving the contract is that it can accommodate incremental change. For example, when the HIL is upgraded to accommodate new hardware, if it adheres to the behaviours within the contracts guarantees (and produces no new application rely conditions) then there is no need to re-examine the applications. Similarly, when an application is altered, as long its behaviour is still consistent with the contract then no alteration is needed to the HIL. One of the next stages of this work is to derive a process for rigorous assessment that a contract is still met following a change.

The discussion in section 6 illustrated areas where there was a need for the client and supplier to negotiate. In this particular instance negotiation was possible as the kernel

¹ Note that this judgement has been made based on this specific system design, and it may be the case that the lack of memory sharing facilities means the HIL is not suitable for another IMA system.

implementation was not yet complete. However, if a Commercial Off The Shelf (COTS) component were to be used there would be no room for negotiation (it would provide "obligation" constraints). It is still important to capture a contract for a COTS product. This helps to determine what other, more flexible components need to do in order to ensure system safety.

The specification of the quantification level of the contract requires some thought when considering incremental change. This level captures details such as response times, the restriction on memory size for the real-time address spaces, and also the reliability of the processing hardware. This sort of data is clearly required for any certification process; however, these details are extremely hardware specific and non-negotiable. In addition, it is unlikely that these details will remain consistent should a new processor be used. In order to try and maintain support for incremental change, where possible the details in the quantifiable level can be given as ranges rather than specific values. For example, the probability in Table 3 could be expressed for the HIL as being at least 10^{-9} . This gives scope for incremental change minimising the potential impact on the applications.

One potential problem with the contract approach is scalability. For a large system with many hazards, a large number of contracts may be derived. This is partially mitigated by fact that the supplier side of the contract (particularly in the case of the HIL) can be re-used by different clients. It may be, however, that contracts should only be developed for those DSRs which address hazards with severe consequences. In order to assess this, a thorough approach to traceability is required. This approach should allow the path of the breakdown of high level system hazards to low level contracts to be easily followed. This would also support assessment of the impact of changes to high level system requirements at the lower levels of the system.

8 Conclusions

This paper has described a modularised safety analysis process which can be used for IMA systems. This process derives a "contract" which can be negotiated between the providers of a supplier component and a client component. The process has been applied to some HIL software proposed for use in an IMA supported navigational display system.

The contract is captured in a manner which supports incremental change by separating out supplier and client guarantees. The analysis technique is being further developed to include guidance for assessing the impact of an incremental change. In addition, further work is required to provide traceability from high level system hazards to low level guarantees.

9 Acknowledgments

This work was undertaken as part of the work of the BAE SYSTEMS funded Dependable Computer Systems Centre (DCSC) by Philippa Conmy and as part of the MATISSE project (EPSRC grant no. GR/R70590/01) by

Mark Nicholson. We would like to thank these organizations for their support.

10 References

- ARINC (1997). *Avionics Application Software Standard Interface ARINC 653*, Aeronautical Radio Inc.
- Bennett, M. and N. Audsley (2001). *Predictable and Efficient Virtual Addressing for Safety Critical Real-Time Systems*. 13th Euromicro Conference on Real-Time Systems.
- Conmy, P., M. Nicholson, et al. (August 2003). *Identifying Safety Dependencies in Modular Computer Systems*. International System Safety Conference, Ottawa, Canada.
- Grigg, A. and N. C. Audsley (1999). *Towards a scheduling and timing analysis solution for integrated modular avionics systems*. *Microprocessors and Microsystems* **22**: 423-431.
- Heiser, G. (2001). *Inside L4/MIPS, Anatomy of a High-Performance Microkernel*, UNSW Technical Report.
- Jones, C. B. (1983). *Specification and Design of (Parallel) Programs*. IFIP Information Processing.
- Meyer, B. (1992). *Applying Design By Contract*. Computer.
- Multedo, G., D. Jibb, et al. (1999). *ASAAC Phase II programme progress on the definition of standards for the core processing architecture of future military aircraft*. *Microprocessors and Microsystems* **23**: 393-407.