Safe Composition of Real Time Software

Iain Bate and Philippa Conmy

Department of Computer Science,
University of York, York,
YO10 5DD, United Kingdom
e-mail: {iain.bate,philippa.conmy}@cs.york.ac.uk

Abstract

There is an increasing move towards the use of modular approaches to software design and implementation in the development of critical systems. The reason is the approaches have a number of benefits including providing support for concurrent development and helping to simplify software maintenance. However, there is little guidance on how to perform a modular safety process for the certification of critical systems as most of the standards assume a monolithic design. Of particular concern is performing safety analyses, with the limited context afforded by a modular approach, in order to derive valid safety requirements with appropriate context / assumptions.

Expressing requirements using contracts is one way to help support change. An example use of contracts between a Real-Time Operating System (RTOS) and application is given. This example has been chosen as the use of an RTOS is an increasingly prevalent form of modularisation, instead of embedding operating system services within the applications. In fact having an RTOS is considered a key enabling technology as it provides a clear interface between the application and platform.

1 Introduction

The need for modular design in systems and software engineering has long been established. The reasons for moving to modular systems include partitioning designs such that different teams can work concurrently and so that change can be handled more effectively. More recently in the critical systems domain, initiatives such as Integrated Modular Avionic systems [7] have tried to introduce the concept to the aerospace domain. Our work [3, 6] has established the importance of contracts between modules for supporting incremental certification and reducing obsolescence. These contracts need to represent not just functionality but also other properties related to dependability.

Laprie [12] originally defined dependability as availability, reliability, maintainability, safety, confidentiality and integrity. Our work has focussed on establishing a method for deriving “safety” requirements based on the failure analysis of components. The requirements are then combined with other information (e.g. context and assumptions) to form contracts. Context describes the type of system the component is operating in, e.g. its operational environment. Assumptions concern the behaviour of other components on the system, e.g. non-interference. These contracts are then used as part of an integration activity. The concepts are general purpose in nature but have been principally developed for the interface between Real-Time Operating Systems (RTOS) and applications. Our previous work has shown how to derive safety requirements for the RTOS [6] and applications independently [3]. In this paper the work is extended and it is shown how the two bodies of work can be integrated.

This paper contributes an approach to the independent safety analysis of an RTOS as part of a modular safety lifecycle. The results of the RTOS analysis is a set of "safety assurance contracts" which are instantiated and tailored to a specific software application. The contracts and evidence that the contracts are met by the system can then be traced back to individually identified software failures.

This paper is laid out as follows. Section 2 provides background on the trends in the use of RTOS, especially in critical systems, their typical features and a discussion of the general problems the features cause for the certification and development of safety critical systems. This leads into an overview, section 3, of a high level process intended to solve many of these problems based on the use of safety "contracts". Section 4 then provides further details on the failure analysis part of the method along with examples of its use. Next, section 5 describes some typical timing features of control systems, and in order to show the types of evidence required. It is then shown in section 6 how contracts between the application and RTOS can be derived. Section 7 provides a discussion of the applicability and limitations associated with the method based on experience applying the method in industry. Finally the conclusions are given in section 8.

2 Motivation and Background

2.1 Using an RTOS in a Safety Critical System

Traditionally, the software in embedded control systems has been written as a monolithic (defined as being composed of a single entity) set of code, tightly coupled to the underlying hardware. This design approach was necessary to ensure that the maximum performance was

achieved and hence timing requirements met. Unfortunately, this type of design means it is very difficult to maintain the software, both to fix application errors or to port the software to a new processor. Some industries, such as avionics, have system lifecycles of 10 or more years, hence hardware obsolescence is a significant problem. This means there is a need to port applications to new hardware at some point in the system's overall lifecycle. In many systems this has effectively lead to a complete re-design as part of the Mid-Life Update [7].

In recent years hardware performance has improved considerably. This helps support a move towards using Real-Time Operating Systems (RTOS) and modular designs whilst still achieving the system timing requirements. By using an RTOS an application no longer needs to contain hardware specific code, making it smaller and easier to maintain. This clearer partitioning of hardware specific code can, with suitable techniques, make porting an application to new hardware easier as the extent of the necessary changes is reduced.

Tanenbaum [1] describes an Operating System (OS) as having two main functions. Firstly, an OS presents the user with a "virtual machine that is easier to program than the underlying hardware". This is usually achieved using an Application Programming Interface (API). Secondly an OS provides "for an orderly and controlled allocation of the processors, memories, and I/O devices among the various programs competing for them". This is particularly important for critical systems.

The typical architecture of an embedded system using an RTOS is a three layer stack as shown in Figure 1. The architecture provides clear abstractions between different parts of the application(s), the OS and the hardware.

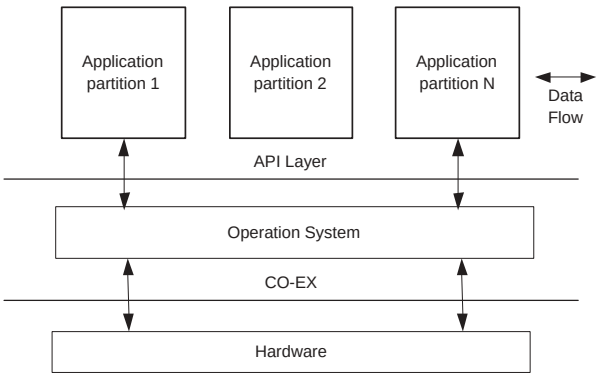

Figure 1. Three Layer Stack Architecture

2.2 Certification Issues and Existing Work

Almost all safety critical systems must go through a certification process prior to their use. As part of this process the system developers must demonstrate that the risk associated with any system hazards (potential to cause harm) is at an acceptable level, and that the system adequately meets its requirements, including timing requirements. This is done by the presentation of a system safety case supported by concrete evidence, i.e. direct evidence produced during design and evaluation. Processes for deriving this evidence are given in standards and guidance documents, which are selected dependent on the domain. For example, UK military equipment developed in recent years has been certified according to Defence Standard 00-56 [15] Issue 2, civil avionics software is generally certified according to DO-178B [21] and IEC 61508 [5] represents a more general safety standard. These standards were written assuming a monolithic software design hence they do not take into account the modular architecture of software supported by an RTOS, but instead examine software and/or system as a whole.

Using a modular development approach based on a RTOS may simplify development and maintenance, but it complicates the process of certification. Firstly, safety analysis is hard to perform in a modular fashion as it relies on whole system knowledge - context, assumptions and most importantly hazards. Secondly, modular approaches place a great deal of responsibility on the integration activities associated with the system.

Some software systems supported by RTOS have been certified using the existing standards. For example, Greenhills Software Inc. have had Integrity 178B certified as part of a high integrity system in a helicopter [9] and the Digital Engine Operating System (DEOS) developed by Honeywell has been certified as part of an aircraft system [18]. The approach taken by these companies (and others) has been to apply some of the procedures of existing standards to the software (such as condition coverage testing to demonstrate each conditional statement can be reached), and then, after integration, perform more testing and analysis that the RTOS works as required in a given context. By using this method it is possible to demonstrate that a particular configuration of the application and RTOS complies to standards so in some ways it can be judged to be successful. But there are a number of problems with forming evidence in this way independent of whether a modular approach is used or not:

1. Firstly, very early on during system development a developer needs to commit to a particular RTOS as the testing and analysis needs to be in a specific environment (performing analysis for an application with multiple RTOS would be extremely costly). It is conceivable that a fundamental problem with the behaviour of the RTOS may only be discovered after the integration testing, at which point a large amount of money and development time has already been spent.

2. Secondly, by tying the evidence to a particular configuration of the system a set of monolithic evidence is produced. This can be extremely difficult to maintain - providing a costly barrier to maintenance.

3. Thirdly, and potentially the most important in the context of this work, there is a problem with the

nature of the evidence itself. Compliance to procedures does not necessarily indicate that a product is fit for purpose. Using the highest level of rigor for testing and verification does in practice lead to the production of a product which meets its specification, but knowing whether that behaviour is safe for a given situation is essential. That is, the requirements have to be validated in the context of intended use.

4. Finally, during the integration of components emergent properties evolve that need to be clearly identified and demonstrated. Emergent properties are especially difficult to handle as a change anywhere in the system can lead to their characteristics being altered.

The authors assert in this paper that the most desirable approach is to perform safety analysis of the RTOS independently of any particular application, and then use the results of the analysis to determine whether a particular RTOS is suitable for a system (allowing the application developers to compare different RTOS). In addition, by performing analysis that identifies particular safety concerns RTOS vendors can develop their product to avoid certain failures, and application developers will be able to develop their software to avoid any identified problems.

Performing analysis independently also has potential pitfalls though. A recent report commissioned by the FAA [10] looked at multiple independent testing and analysis techniques for RTOS (e.g. fault injection and formal analysis) and concluded that without context it was extremely difficult to demonstrate that the results are of relevance to any particular application.

Safety and timing are emergent system properties [13] and because of this integrated components don't always behave as expected even if they individually behave as specified. One famous example of an RTOS problem is the Mars pathfinder mission [8] which started to suffer from persistent resets a few days into its mission, resulting in the loss of the system for considerable periods of time. After some testing and analysis it was discovered that the software fault was caused by priority inversion. A low priority software task on board the pathfinder was sharing a resource with a high priority task and the low priority task blocked this resource after it was preempted by other medium priority tasks. When another high priority task discovered the previous high priority task had not completed, it initiated a system reset. A global, default, setting in the on board RTOS was allowing the priority inversion to take place. So although the RTOS was performing as intended, its behaviour was wrong in context due to an emergent property.

Therefore any approach taken for independent safety analysis of RTOS must provide adequate coverage of the range and types of failures which could lead to system level hazards, and must also support a way to link those failures to hazards.

3 Overview of The Method

This section provides an overview of the approach proposed in this paper and supporting safety processes presented. The approach chosen is to allow engineers to perform analysis of the RTOS early in the system process and then develop an application using the results.

The lifecycle and related processes of the approach are summarised in Figure 2. Firstly, the RTOS is analysed (see the LHS) and template contracts are derived which describe the conditions under which certain failure conditions can be avoided. An application developer will then use these analysis results to determine which failure conditions are relevant or credible for their system, and then instantiate a set of template contracts as part of the trade-off process shown in the centre of the diagram. Finally evidence will be gathered to show the contract is met.

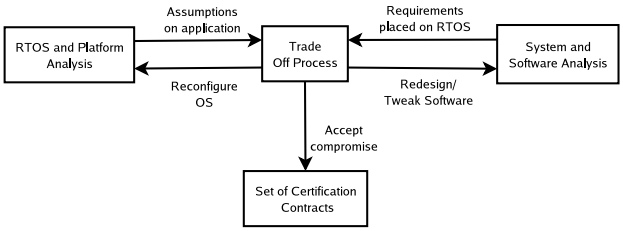

Figure 2. Modular software safety process

The template contracts for the RTOS are derived in two stages. First an adapted failure analysis technique is used to determine potential failures in the interactions between an application and the RTOS. The technique uses guidewords, e.g. omission, which are applied to a function or data flow in order to determine potential failures. The output from this analysis is a set of Derived Requirements (DR) on the RTOS, hardware and applications. The second part of the process is the development of contracts for each of the derived requirements. The contracts contain constraints on various properties of the applications and RTOS, including architectural design (e.g. number of permitted processes), behaviour (e.g. the creation and destruction of processes), and performance characteristics (e.g. execution times). Contracts are developed by examining the RTOS source code.

The next section describes the first part of the process in more detail, using an example for illustration. The second part of the process is described in section 6.

4 Performing Failure Analysis of an RTOS

This section provides an overview of the method used for the failure analysis of the RTOS and demonstrates its application to a small microkernel "L4" [11]. This example was chosen as its small size is ideal for discussion here for space reasons. Other work has looked at the application of the failure analysis technique to more mainstream OS such as Linux [6, 19].

The approach taken is to adapt an existing software safety analysis technique - SHARD [20]. The advantage of this approach is that safety engineers are familiar with

this type of process which helps increase the confidence in the results they produce. By failure we mean any deviation from the required intent e.g. a thread starting execution later than intended. This includes systematic errors (i.e. design faults) as well as any random failures in the supporting hardware. These failures may lead to hazards in a given situation e.g. the late response of the software controlling a braking system may be hazardous.

Initial attempts at analysis looked at each API call in isolation, but the results were not that informative as it wasn't clear what the effect of the failure was, or how it could be caused. Therefore a framework has been developed which identifies the scenarios in which different API calls are made. This is described in the following subsection and then examples of its use presented in subsequent sub-sections.

4.1 Summary of the Analysis Process

The analysis process follows the five steps given below. The basis for the process is to first establish an understanding of the RTOS and its services / calls, then determine how the RTOS fails so that suitable derived requirements can be produced.

1. The analyser familiarises themselves with the RTOS specification. They should in particular look at communications protocols, scheduling mechanisms (e.g. prioritisation methods), and memory management design.

2. The analyser lists all the Application Programming Interface (API) calls, along with other key services provided by the RTOS. The latter list would typically include scheduling and memory management but may also include other services such as device drivers.

3. The list of services should be matched against the 6 high level functions provided in table 2, using a matrix as shown. The purpose of this exercise is to assess the range of different scenarios the API calls and services are used in, and hence help to ensure completeness in the analysis. The 6 functions have been established and found to be sufficient for a number of different OS analyses (the reader should refer to [6, 19] for more details).

4. The analyser performs a failure analysis for each of the functions in turn. This analysis is an adaptation of the Software Hazard and Resolution in Design (SHARD) [20] analysis which uses guidewords (omission, commission, early, late, value) to suggest possible failure modes. The reason a guideword-based approach has been chosen is to further help ensure completeness (these guidewords have been demonstrated to categorise all software failures - see [20]). The columns used in the analysis are shown in table 3, and the method is summarised as follows:

   (a) The analyser first lists all the failures which are implied by the guidewords

   (b) Then for each failure the assessor considers how they could arise, in particular they should consider how misuse of each of the mapped services could cause that failure. Often there are multiple causes of failures. These should all be listed.

   (c) The analyser should suggest DR which could be used to prevent the listed causes of failures.

5. Finally, the analyser consolidates the DR, as inevitably there are repetitions in proposed solutions.

4.2 Example of Mapping Using L4

This section illustrates the application of the mapping the list of services (stage 3 of the process) to a version of the L4 microkernel [11]. The particular version used was modified to support high-integrity avionics software, within a distributed system [4]. The main advantages of using this microkernel are that the authors had access to the source code and that it is a small well written RTOS. The core L4 philosophy is the provision of only the most basic RTOS services and that the RTOS software runs in privileged mode on the processor. Other software (including an additional RTOS services if required) is then implemented as a set of threads running in user mode. This architecture is extremely flexible and secure. The microkernel has only 7 API calls, and these are summarised as follows:

- **task_new** - task creation and deletion. A task is an address space within which a number of different threads run (usually fixed at 128).

- **thread_switch** - releases the processor so that another thread can consume processor time. Caller may specify a thread to switch to or just donate time to next scheduled thread.

- **lthread_ex_regs** - reads and writes the register values of a thread in the current task

- **lthread_schedule** - used to set the priority, time/-slice length, and externel pre-empter of other threads. Also allows thread states to be retrieved.

- **IPC** - Inter Process Communication, unbuffered and synchronous communications between threads are supported. IPC calls allow timeouts to be specified on IPC waiting in order to prevent blocking.

- **id_nearest** - used to provide thread identifiers during IPC, or for setting up registers for a thread.

- **fpage_unmap** - used to unmap the specified memory page from all address spaces into which the invoker mapped it

It should be noted the clans and chiefs protocol (refer to [11] for further details) often associated with L4 was not used due to the performance overhead associated with the protocol. The microkernel contained special "real-time address spaces" with predictable memory access times as well as normal virtual addressing. Fixed priority scheduling was used with round robin scheduling to arbitrate between runnable tasks that share the same priority.

The results of the service to function mapping is shown in table 2. The services shown are the 7 API calls and also memory management and scheduling. The results show that each of the services are used to support multiple high level functions. For example, the *thread_switch* call is required during communications in order to allow data to be sent, is used to allow other threads (including health monitor threads) access to the processor, and also to support a consistent execution environment.

4.3 Example Failure Analysis

Table 3 shows an extract of the failure analysis of the second function, $thread\_switch$, which produces controlled access to processing resources. Examination of the results reveals a mixture of derived requirements. Some requirements are placed on the application (e.g. to perform analysis to ensure correct priorities are assigned). Some are placed on the RTOS (e.g. to ensure the microkernel does not allow memory corruption to change priorities). Some requirements are also placed on the hardware (e.g. to ensure random corruption is sufficiently low). Many of the DR have options over what is specified. Where there is an option one or both requirements can be used. The decision over which option(s) to take would be part of the overall design trade-off process. Refer to section 6.4 for further details. Other DR require multiple conditions to be satisfied. These results indicate that the DR must be allowed for during application development if failures are to be avoided. Section 6 describes the next phase of our RTOS analysis which derives contracts for the integration process.

One thing not immediately obvious from the extract shown is that many of the derived requirements are repeated. For example, many of the omission failures have related commission failures and the same DR can be used to protect against them. An instance of this is the release of tasks at the correct rate which in the presence of failures can lead to too many or too few tasks being released. In our experience of applying this technique the number of consolidated requirements is usually between 20 and 30 for a component of this size and complexity. This means that although the failure analysis tables can be large, the actual consolidated results are manageable. Another point of note from these results is that one derived requirement states that interrupts should be disabled and another that they should be enforced. Which DR is needed will depend on whether late failures are deemed critical for a specific context. Again, this is discussed in section 6. However, first we look at timing aspects from an application perspective in order to assess whether the DR are relevant to application development.

5 Timing Requirements Derived from the Applications

This section describes how application timing requirements are derived for control based systems, in order to provide a baseline for the assessment of the relevance of the RTOS analysis results.

5.1 Timing Requirements

All scheduling approaches require a minimum set of information about timing requirements so that an appropriate scheduler can be produced. For most scheduling approaches the minimum set of information is the deadline and period of tasks. However for more complex systems, e.g. those that feature task dependencies, more information is needed. Previous work [2, 22] has demonstrated that the typical timing requirements of applications can be represented by the following.

1. *Independent tasks and messages*
   (a) *period* - the rate at which a periodic task or message is to be executed at.
   (b) *deadline* - the time from release until when a task or message has to complete.
   (c) *completion jitter* - the allowed variation in task or message completion.
2. *Dependencies*
   (a) *precedence* - the order in which tasks and messages should be executed. A precedence sequence is sometimes referred to as a transaction or process chain.
   (b) *end-to-end deadline* - the allowed time from the release of the first task to the completion of the last task in a particular precedence sequence.
   (c) *separation* - the minimum time between a task or message commencing its execution from the completion of an earlier task or message in the precedence sequence.
   (d) *completion jitter* - the allowed variation in a precedence sequence completing.

The timing requirements can be explained in the context of a typical control system. For this purpose, a simple Proportional Integral Differential (PID) loop is used.

5.2 PID Loop

The main purpose of a PID loop is to ensure a sufficiently fast response to inputs whilst maintaining stability, accuracy and limits on data. Figure 3 depicts a typical PID loop used to control the operation of a plant.

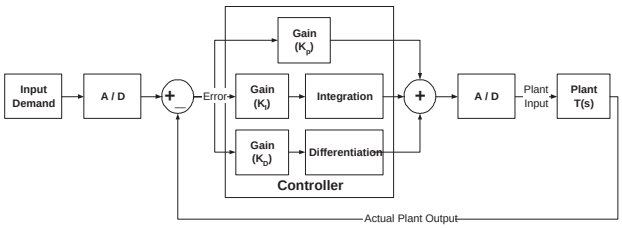

Figure 3. PID Loop

In its simplest form, a continuous ideal domain representation, the output of the PID loop is the plant input. The control system input is the difference between the input demand, which is the desired plant state, and the plant's actual output and it is referred to as the error.

In a computer-based implementation of the PID loop, the Input Demand (e.g. pilot stick position) and the Actual Plant Output (e.g. aircraft's flap position) are usually analogue signals. The computer performs the rest of the processing in the digital domain. Converters (A/D) are used to sample the analogue signals, e.g. to produce the Error input, and then converted back to analogue values at the output. Converting back to an analogue signal is often referred to as digital to analogue conversion (D/A), de-sampling or actuation. In order to give better control over jitter, the functionality that needs to be performed in software is normally split into three separate tasks - sampling, calculation and actuation [2, 22]. The tasks can be modelled as either periodics or sporadics.

5.3 Scheduling Properties

It is, of course, essential that the sampling, core functions and de-sampling tasks are executed in that order. Other work, e.g. [2], has shown how to specify and control the precedence of functionality for a PID loop to ensure that these requirements are met. Figure 4 presents properties for a typical transaction of a control loop that can be controlled by the scheduler. As stated earlier, the three tasks are sampling of sensor data, calculation and actuation output. The Figure shows:

1. how each task has jitter comprising both release and execution jitter as well as an invariant in its execution time,

2. there is jitter on both sensor capture (referred to as sampling jitter) and actuation (referred to as de-sampling jitter),

3. a task must be completed before the next task in the transaction starts its execution so that the next task can use fresh data,

4. the response time of a transaction is equal to the time between the release of the first task and the completion of the last task (the worst-case response time for a transaction must be less than its deadline), and

5. the period of a task is the time between two consecutive earliest releases.

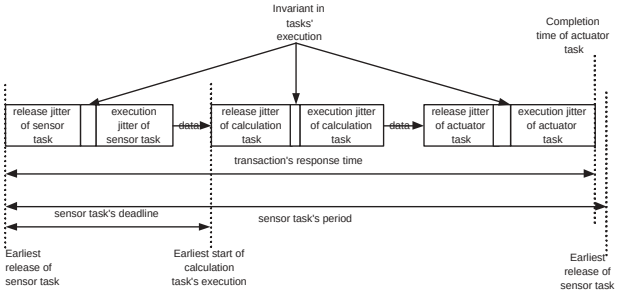

Figure 4. Scheduling Properties

5.4 Other Factors

Related to the application, there are other requirements that emerge. The principal ones are for validating the correct operation of the system, often referred to as health monitoring. The purpose of this is three fold:

1. *Sensor* - Ensure that potentially erroneous input (sensor) data is not used and hence polluting the calculated value. An advantage of the PID loop is that any incorrect data has limited effect due to the differentiation smoothing some of its effect away. However there is the disadvantage that the feedback will mean any effect lasts for more than one sample, although with diminishing effects.

2. *Actuation* - Ensure that incorrect, and potentially damaging, data is not output. An example of this form of output is data that is out of range which could lead to an actuator being driven beyond its limits and result in mechanical wear or damage.

3. *Calculation* - Determine whether the appropriate (sufficiently accurate) translations are being performed from input to output.

It should be noted that the three requirements relate directly to the three phases of computation shown in section 5.2. These requirements are often handled through validation checks.

5.5 Requirements Placed on the Architecture

Based on the contents of this section, the following is a summary of the application requirements on the underlying platform including RTOS. The text in *italics* shows how the results of the RTOS failure analysis process, given in Table 3, has helped identify potential failures in supporting each of the requirements. Hence traceable links between the analysis and derived requirements of the RTOS and application are formed.

1. *Req 1* - Tasks are released correctly. Periodic tasks are released at the correct time and rate. Sporadic tasks are released upon the appropriate event occurring. *The failure analysis uncovered a number of potential causes of failures in task release, e.g. C1, C2 and C6.*

2. *Req 2* - Tasks are released in the correct order. For example in a priority driven scheduler, tasks are released in priority order. If the priorities are dynamically calculated, then there is also a requirement for this. *Again a number of potential failures related to priorities are identified e.g. C4 and C5.*

3. *Req 3* - Tasks do not suffer too much release jitter. This places requirements on both the task release mechanism and the task attributes assigned. *There are multiple failures in the "late" section of the analysis related to release jitter e.g. C6 and C7. It should be noted that the full analysis would also result in failures related to early release of tasks that also affects jitter.*

4. *Req 4* - Tasks do not suffer too much execution jitter. This places requirements on the processor (execution times need to be predictable and within acceptable bounds), the task release mechanism and the task attributes assigned. *Again, there are multiple failures related to this e.g. C6, C7 and C8.*

5. *Req 5* - Tasks complete by their deadline. This places requirements on the processor (execution times need to be predictable and within acceptable bounds), the scheduling mechanism and the task attributes assigned. *A good example of a related failure is C8 where poor memory management could lead to extended execution times.*

6. *Req 6* - Tasks have to be executed in the correct order. This places requirements on the scheduling mechanism and task attributes assigned. *Many of the omission/commission failures related to this (e.g. C4 and C5)*

7. *Req 7* - Tasks have to receive correct data. This places requirements on health monitoring and the communications. *Analysis of functions 1 and 4 in table 2 uncovered many failures of concern to this requirement. There is also a link to early and late failures (e.g. C6, C7, and C8) as any continuous signal that is sampled early and late will have different values to those expected [2, 22].*

8. *Req 8* - Tasks have to convert data appropriately. This places requirements on the software design but also on whether the source code intent is upheld. *Analysis of this requirement generally relates to ensuring the application behaves as intended, however analysis of function 5 in table 2 was revealing as it concerned whether correct versions of tasks are loaded and executed.*

6 Generation and Use of Contracts

In section 5.5 we showed that failures relevant to application development were found by the analysis process. But in section 2.2 we stated that it was necessary not only to uncover relevant failures but also to provide traceability from these to system hazards. This section describes a process developed to support traceability, and also provide detailed evidence that derived requirements placed on the RTOS have been addressed. The process revolves around the production of a set of "safety assurance contracts" which describe the conditions under which the RTOS' DR will be met.

6.1 Generating Safety Assurance Contracts

The use of contracts is a well-known approach to support software composition [14, 17]. In this approach each software function has a set of pre and post conditions at its interface which constrain the relationship between a client piece of software calling a supplier function. The post-conditions describe what the function will ensure if the pre-conditions are met. This relationship is summarised in Figure 5.

Composition is made more predictable since the assumptions and provided behaviour of a component are explicitly stated. This method works well where the supplier is purely a slave to the client. However, the vertical relationships between applications, operating system and hardware are more complex as each has a level of autonomy which is not usually controlled via an interface. For example, an RTOS will decide when a thread can execute based on a scheduling mechanism which may not be under the control of the application layer. Therefore some adaptations to the concept have been made.

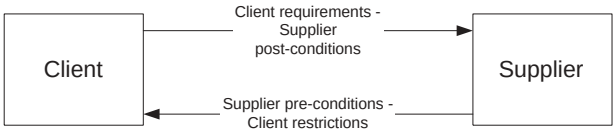

Figure 5. Basic concept of a software contract

6.2 Contracts on the RTOS

This section describes how the contracts are generated for the RTOS. The contracts contain information on how and if the RTOS supports the DR (the RTOS guarantees, i.e. post-conditions). If it does support the DR it is important to identify under what conditions it provides the support. These conditions are "pre-conditions" on the applications usage of the RTOS.

Deriving the content for a "safety assurance contract" requires access to the RTOS source code. This is due to the need to identify all relevant code fragments. For example, code which manipulates access to memory is often found in multiple sections of the RTOS. All these sections must be examined for any behaviour relevant to a memory DR. As a result either the RTOS developer must perform this analysis, or an agreement must be entered into with the RTOS provider to view the source code.

The contract generated is a template only, in that all that is described is what the RTOS offers. The acceptability of

the contract, and its conditions, will need to be assessed on a case by case basis once a specific application is known. For example, the RTOS contract may detail the length of time taken for a communications message to be sent given a certain set of parameters. However, the acceptability of this time can only be determined for a specific application. The time can be verified both off-line and on-line. The on-line options include performing checks within the RTOS (requiring changes to the current RTOS), within a wrapper interface around the RTOS or within the application.

The contract is formed by considering three key sets of properties - process architecture, behaviour and performance. Constraints upon these form the pre and post conditions.

The following summarises the process for contract generation:

1. The analyser should familiarise themselves with the code architecture of the RTOS and identify the location of key software such as memory management and scheduling.

2. The analyser should identify and produce a draft list (which is finalised during the trade-off analysis process) of architectural constraints and guarantees.

3. The analyser should identify and produce a draft list of behavioural constraints and guarantees, examining both API calls and also scheduling mechanisms.

4. The analyser should identify and produce a draft list of performance constraints and guarantees, examining execution times for API calls, context switches and internal RTOS calls, list any known memory constraints, and list any size limitations. These lists are used as a basis for contract constraints.

5. Then for each DR placed on the RTOS the analyser should:

   (a) Identify constraints relevant to DR.

   (b) Trade off and link constraints where relevant for each of the constraint types. For example:

   - performance versus architecture - e.g. speed of communications may depend on whether data is sent either to an independent memory space on the same processor or to a separate processor.
   - behaviour versus performance - e.g. bit parity checking on communications may impact its speed but has the benefit that it helps increase the system's fault tolerance.
   - behaviour versus architecture - e.g. one thread cannot control the execution of another thread unless it is its parent.

   (c) Mark each resultant constraint as either Fixed (i.e. the application must adhere to it) or Choice (where two or more alternatives exist or if there is a chance to alter the RTOS code or configuration for a particular constraint).

   (d) Store results as set of RTOS Guarantee vs Application Constraints as shown in Table 1.

RTOS Guarantee	Application Constraint	Choice / Fixed
Each virtual address is mapped directly to a physical address ensuring predictable access times - $Req8$.	Tasks address spaces must be no greater than 256K in size.	Fixed

Table 1. Example contract for real-time address space constraints

6.3 Contract Example

One requirement, DR8 in Table 3 was to use the real-time address spaces to ensure threads have deterministic execution times. Examination of the code revealed a restriction on the size of tasks created in real-time address spaces. By using ring fenced memory areas fixed memory access times are achieved. This leads to a limitation on tasks' addresses space of 256 kbytes. This means some of the larger tasks may need to be split to work as expected with the L4 kernel. With respect to the PID loop example, the calculation task is the one that is likely to use more memory - although 256 kbytes should be adequate in practice. This task could easily be split into four calculation component parts - combination, proportional component, integration and differentiation. However it is likely that it is the health monitoring functionality associated with the calculation that uses more address space so a more logical split might be to have two tasks - calculation and health monitoring. This split may increase the amount of data flow (which is a performance disadvantage) but gives better partitioning (which is a safety advantage).

An example contract showing the relationship between application and RTOS is shown in table 1. The choice / fixed column refers to situations where there may be a choice in the ways that the contract can be satisfied.

6.4 Trade-off Process

The final part of the RTOS analysis is to integrate the results and instantiate the contracts as necessary. For this the application developer has to revisit the failure tables from the initial RTOS analysis and add an extra column for listing the effect of the failure, and whether it is hazardous. For reasons of space this is not shown in this paper.

Using the flight control PID loop example as described in section 5.2. A failure leading to a thread completing later than required would clearly be hazardous for flight control of aerodynamically unstable aircraft, therefore preventing these failures would be extremely important. However for an engine control system a late value

may be less of a problem as the data is mainly used to make adjustments to fuel flow to respond to pilot requests or correct mechanical drift. That is, a late value will only mean that the fuel flow rate is in error for a short period of time and not potentially lead to the loss of the aircraft as might be the case for flight control systems.

The DR contract for predictable memory access times needs to be instantiated, by ensuring the constraint on task sizes could be met for each of the three tasks. Where the constraints is not easily met, then there are a number of trade-offs that can be made. The options include; splitting tasks so that off-line analysis can ensure the constraint is not broken, using exception handlers to identify on-line when the constraint is broken and taking an appropriate action, or even changing / re-designing the RTOS. Each of the options has implications. For instance, splitting tasks will mean the system has more tasks and hence greater overheads in general. Employing exception handlers would add complexity to the design and safety analysis of the system. However it would not affect the system's overheads in the general case (only in overload conditions which may be infrequent or justified to be improbable) and it would mean the logical design of the software is not affected by the choice of the RTOS.

By linking generic failures of the initial analysis to specific events and then instantiating related contracts, improved traceability from failures to hazards and then to evidence is achieved.

7 Discussion

This section presents discussion on some of the issues raised by our research and experience gained through its application by industry.

7.1 Scalability

A potential problem with any contract-based approach is scalability. The RTOS and applications examined in this paper are relatively small, but have still generated large tables of data. However not all of this data is relevant to a particular application as analysis of all RTOS calls is performed as the results are intended to be used for multiple applications. Whilst scalability is a concern, with judicious application of the method in certain key areas of the system, e.g. to the RTOS / application interface, 'real' benefit can be provided. In addition work is on-going on methods to cut down the amount of data generated by looking for patterns and identifying cut-sets. The drawbacks are more than compensated by the provision of support for incremental certification rather than having whole system change as currently performed in a mid-life update. Currently industry are using the method and the results will be reported in later work.

7.2 New Safety Standards

Another potential issue is that our technique is not compatible with certification practices. However, Issue 3 of Def-Stan 00-56 [16], has been written with a much more flexible approach, allowing the applicant to use new methods of analysis as long as they provide adequate evidence that the system is acceptably safe. In addition the new civil aviation standard for modular avionics systems [23], which at time of writing is undergoing final editing, suggests using "tiers of integration" for staged analysis and development which is also compatible with our approach.

7.3 Managing Change

In section 2.1 simplifying maintenance as well as developing software in a modular way was discussed. Theoretically if the RTOS or applications are altered, as long as the software still conforms to the safety assurance contract conditions, then the change has no impact on system safety. In practice, however, the large number of dependencies between the constraints means this is difficult to achieve. However difficult it is to achieve, the method has the considerable benefit of providing an understanding of the impact of change. This should help contain changes and help reduce regression testing. A related benefit is that the method gives a significant step forward in the ability to support parallel development and manage emergent properties that occur when components are combined.

7.4 Completeness

A final issue is that of completeness. Whilst the RTOS analysis technique has been developed to attempt to maximise coverage of both failure conditions and usage scenarios, actually demonstrating that every potential failure has been uncovered is impossible since each application will be different. This criticism could be levelled at every form of requirements capture though.

8 Conclusions

This paper has presented an approach for the independent safety analysis of an RTOS and shown how the results can be used as part of a modular safety process, using a simple microkernel as an exemplar. Further research is being undertaken to improve our technique, in particular looking at scalability and how the derived requirements on the applications and hardware can be enforced.

References

[1] A.S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, 2nd edition, 2001.

[2] I. Bate and A. Burns. An integrated approach to scheduling in safety-critical embedded control systems. *Real-Time Systems Journal*, 25(1):5–37, Jul 2003.

[3] I. Bate and T. Kelly. Architectural considerations in the certification of modular systems. *Reliability Engineering and System Safety*, 81:303–324, 2003.

[4] M. D. Bennett and N. C. Audsley. Developing a real-time microkernel design process. *Proceedings Work in Progress, Real-Time Systems Symposium*, 2001.

[5] CENELEC. *IEC 61508 Functional Safety of electrical/electronic/programmable electronic safety-related systems.* 2001.

[6] P. Conmy, J. McDermid, and M. Nicholson. Safety assurance contracts for integrated modular avionics. In *8th Australian Workshop on Industrial Experience with Safety Critical Systems and Software*, October 2003.

[7] R. A. Edwards. ASAAC phase I harmonized concept summary. In *Proceedings ERA Avionics Conference and Exhibition*, London, UK, 1994.

[8] G. Reeves and Mars Pathfinder Flight Software Cognizant Engineer. What really happened on Mars ? - Authoritative Account. URL:http://research.microsoft.com/ mbj/Mars_Pathfinder/Authoritative_Account.html, December 1997.

[9] Greenhills. FAA Certifies INTEGRITY RTOS for DO-178B Level A Use In Sikorsky S-92 Helicopter. http://www.ghs.com/news/230210r.html, 2003.

[10] V. Halwan and J. Krodel. Study of Commercial Off the Shelf (COTS) Real-Time Operating Systems (RTOS) in Aviation Applications. Technical Report DOT/FAA/AR-02/118, Federal Aviation Authority, December 2002.

[11] G. Heiser. *Inside L4/MIPS, Anatomy of a High-Performance Microkernel*. Number UNSW Technical Report. 2001.

[12] J. Laprie, J. Arlat, C. Beounes, K. Kanoun, and C. Hourtelle. Hardware and software fault-tolerance: Definition and analysis of architectural solutions. *7th Annual International Symposium Fault-Tolerant Computing*, pages 116–121, 1987.

[13] N. G. Leveson. *Safeware*. Addison-Wesley, 1995.

[14] B. Meyer. Applying design by contract. In *Computer*, pages 40–51, 1992.

[15] Ministry of Defence. *Safety Management Requirements for Defence Systems, DEF-STAN 00- 56, Issue 2*. 1996.

[16] Ministry of Defence. *Safety Management Requirements for Defence Systems, DEF-STAN 00- 56, Draft Issue 3*. 2004.

[17] Object Management Group. *Response to the UML 2.0 OCL RFP, Submission Ver 1.6*. January 2003.

[18] J. Penix, W. Visser, E. Engstrom, A. Larson, and N. Weininger. Verification of time partitioning in the DEOS scheduler kernel. In *International Conference on Software Engineering*, pages 488–497, 2000.

[19] R. Pierce, M. Nicholson, and A. Faulkner. Assessing operating systems for safety related applications. In *Proceedings of International Systems Safety Conference*, Ottawa, Canada, 2003.

[20] D. Pumfrey. *The Principled Design of Computer System Safety Analyses*. PhD thesis, Department of Computer Science, University of York, 2000.

[21] RTCA-EUROCAE. Software Considerations In Airborne Systems and Equipment Certification DO-178B/ED-12B. Technical report, RTCA and EUROCAE, 1992.

[22] M. Torngren. Fundamentals of implementing real-time control applications in distributed computer systems. *Real-Time Systems*, 14(3):219–250, May 1998.

[23] WG-60/SC200 Working Group. Modular Avionics. http://www.rtca.org/comm/sc200.asp, 2004.

Function	**task_ new**	**thread_ switch**	**lthread_ ex_regs**	**lthread_ schedule**	**IPC**	**id_ nearest**	**fpage_ unmap**	**Memory Management**	**Scheduling**
1-Secure & timely communications	X	X	X	X	X	X	X	X	X
2-Controlled access to processor	X	X	X	X	X	X	X		X
3-Secure data management					X	X	X	X	
4-Health monitoring	X	X	X	X	X	X	X	X	X
5-Consistent execution environment	X	X	X	X	X	X	X	X	X
6-General access to processing	X		X	X		X		X	X

Table 2. Matrix matching low-level services (columns) to high-level functions (rows)

Guideword	**Failure**	**Cause**	**Derived Requirement**
Omission	Thread does not run when it should have	C1 - thread_switch not called by application	DR1 - Ensure thread scheduling calls are correct
		C2 - thread_switch called but called thread blocked	DR2 - Use IPC timeouts to prevent blocking where necessary AND/OR interrupt used to notify of switch failure
		C3 - Thread has been deleted	DR3 - Ensure L4 does not delete threads and ensure application does not delete threads AND/OR use backup lanes
		C4 - lthread_schedule called with wrong priority for process	DR4 - Ensure application calls with correct priority AND ensure application threads cannot change priorities
		C5 - Thread priority has been altered due to data corruption	DR5 - Ensure random corruption is sufficiently low AND ensure no corruption from application threads not allowed to change priority
Late	Thread starts execution late	C6 - Previous thread has overrun	DR6 - Interrupts enforced for certain threads
	Thread finishes late	C7 - Thread has been interrupted during execution	DR7 - Interrupts disabled for certain threads
		C8 - Thread has taken longer than expected due to memory access times	DR8 - Use real-time address spaces

Table 3. Extract of Analysis of "Controlled Access to Processing Resources"