IST-2001 34140

Final Interface Definition

Deliverable D-AF.2v2

Ian Broster
Alan Burns
Gerhard Fohler
Julio Medina
Michael Gonzalez Harbour

April 2004

Contents

Section 1

Introduction and Rationale

The purpose of this document is to present the final contract-based interface to the FIRST scheduling framework (FSF). It briefly reviews the types of application requirements that are supported and presents the contract itself. This is a second phase document, extending and drawing from D-AF.1-v1, D-AF.1-v2 and D-AF.2-v1 and other FIRST deliverables.

The motivations for a contract-based approach are summarised as follows,

- isolated development of components;
- integration of legacy applications;
- portable applications;
- open systems;
- standard interface.

Key in FIRST is the concept of integration of complex applications with a variety of real-time requirements. In particular, we consider cover a broad range of realistic application requirements and real-time behaviour, including well-understood *hard real-time* behaviour as well as less well defined concepts of *soft* requirements. It is noted that in many modern real-time systems, only a small fraction of the system could be considered to have hard deadlines, yet most of the rest of the system might have important timing or performance requirements.

Previous work presented in D-AF.1-v1, drawing from academia and industry, presented a set of characteristics of modern and future real-time systems. The set of systems that we consider real-time systems (and therefore within the scope of this framework) is very broad, including systems with the most critical timing requirements and systems with vague quality of service ideals. D-AF.1-v2 carefully defined this scope. Based on these characteristics, a list of supported application requirements has been produced.

Research within FIRST and elsewhere indicates that a wide variety of requirements can be supported by *server* technologies supporting a hierarchical scheduling structure. Interface to the servers is provided by a *contract*—an agreement between application and scheduler about the real-time behaviour of the application and the service provided by the scheduler. The essence of the contract is a definition of the bandwidth that the application requires with the CPU.

1.1 Application Requirements

D-AF1v1 promoted the concept that application requirements for timing could be considered in 5 classes:

Periodicity. How *often* some processing needs to be performed;

Resource usage. How *much* processing is required;

Performance and guarantees. Concerning *measurement* and *verification* of the performance and quality of service of the processes;

Change management and maintenance. How *changes* in the real-time characteristics are handled (both off-line and on-line), including composability of systems;

Dependability. Relating to how *important* it is for the above features, particularly performance and guarantees, to be adhered to.

It is apparent that the first three of these (periodicity, resource usage, performance and guarantees) are a generalisation of the well-established terms period (T), worst case execution time (C) and deadline (D) used to describe simple periodic processes.

We also note that the periodic process model can usefully be generalised to a *job model* that is representative of the structure of many components in real-time systems. That is, systems are decomposed into 'activities' where each activity is a piece of functionality that is performed *repeatedly*.

Thus, we may define an application as a set of repeating *jobs* that may be executed in logical parallelism; the frequency, duration and other constraints of the jobs are described using the scheme above.

This scheme must not be interpreted as another implementation of a periodic process model; the motivation for the FIRST project is to be able to support applications that have considerably more *flexible* timing requirements. There is no implication that a description of the *periodicity* of a job is a single, constant value; the FIRST framework allows any given job instance to have a different time between invocations. For example, an elastic period could be used, where the actual inter-arrival time depends on available resources, or the inter-arrival time could be chosen from a set of suitable values for which the application can operate efficiently.

1.2 Flexibility

The ability support *flexible* application timing requirements is at the heart of the FSF. This flexibility can include:

Ability to change. The timing behaviour of an application can change throughout the execution of the software—mode changes, environmental factors and interactions with other applications are some of the reasons why an application's timing characteristics may change. The FSF can support changes through a combination of *contract re-negotiation* and dynamic reclaiming of resources.

Support for unknown or partially known requirements. For many algorithms and software, the the timing behaviour is unknown or partially known. For example: the execution time or worst case execution time (of a job) may not be computable offline, even if either can be calculated, worst case execution time analysis always frequently returns pessimistic results. Some algorithms (anytime algorithms, for example) can usefully take as much CPU time as they are offered; the accuracy of calculation depends on the time taken to compute it. The FSF can support unknown and partially unknown timing requirements through flexible contract support, temporal isolation and efficient resource reclaiming.

Support for use of spare resources. Applications can request to use unused CPU bandwidth. This may be done either by contract re-negotiation (or static reclamation)—where additional bandwidth is allocated to the application until it explicitly drops it, or by dynamic reclamation—where any spare time that emerges at run-time can efficiently be allocated to jobs that can use it.

Temporal partitioning. Support for temporal partitioning allows an application to rely on the operating system to ensure that its behaviour cannot adversely affect the timing of other applications. This is a very useful scheme for many soft applications or applications with unknown timing requirements. It allows the designer to focus on the functional aspect of the behaviour, where appropriate, rather than having to guarantee 'artificial' timing constraints on an algorithm that does not naturally map to a real-time system. Operating system functions are provided to ease integration.

Section 2

The Application View of the Contract

The nature of the contract process is that the application sends a request to the scheduler for a contract. If the scheduler is able to support that contract, then it agrees the contract. Once agreed, the contract is binding and the scheduler will continue to support the contract until the application either deletes the contract or requests to renegotiate it.

2.1 Servers and Bandwidth

The fundamental part of the contract is that the application requests a minimum (and optionally, a maximum) level of 'service' (CPU bandwidth) from the operating system. If this request is agreed then it forms a *contract* between the application and the operating system. The contract guarantees that the application will receive the minimum service. Further, if there is additional spare resource available then the application may be allowed to use this resource until it reaches the maximum requested.

The contract is implemented on an underlying server-based approach, where a server is a basic budgeting resource. The server, i, has parameters C_i, T_i, D_i which are the server-budget, server-cycle (period) and server-deadline. These represent the basic contract. The semantics are that at least C_i CPU time is available every T_i before D_i time has elapsed.

Thus, the view of an application, running in a FIRST framework is that it has a proportion of the CPU bandwidth, as defined by the server parameters, that it uses according to the needs of its processes and the policy of its local scheduler.

Note that C_i, T_i and D_i do not necessarily relate to the parameters of a periodic process—mapping application requirements to these parameters is described elsewhere (in D-AF.1-v2, D-SI.4-v1 and in future deliverables).

A server may be considered as a time-budgeting or accounting mechanism. In abstract, a server is merely a small set of parameters and variables which record the requested and used resources of the server. However, it is not necessary to adhere to this abstraction in an implementation; in an efficient design, a server may also have a functional component. Each server manages the budget for a scheduler, applications or processes.

There are numerous types of possible servers, with different characteristics. Despite the suitability of some of these, in FIRST, a particular type of server is promoted. The server is described in D-SI.4-v1. This server may be considered a general abstraction of a number of standard servers, but with some important differences. The resulting server is similar to a constant bandwidth server (CBS) in its properties and behaviour.

2.2 Acceptance of Contracts

The contract can be negotiated (and re-negotiated) at run-time by the application. Acceptance of the requested contract is not guaranteed. However, for closed systems, off-line analysis can be used to ensure all contracts can be honoured; for such systems, the acceptance test may be disabled.

Once accepted, the operating system is bound to honour existing contracts. In the case of a re-negotiation of a contract, if the new contract cannot be accepted, the application is still guaranteed to receive the previously accepted contract.

The acceptance test is implementation defined. A utilisation based test provides a simple, effective approach. An acceptance test base on response time analysis is also promoted.

Section 3

Modular Contract Design

The contract with the scheduler has a large number of parameters, many of which are optional. The FSF is structured as a number of modules in order to limit the complexity. Each module, except the CORE module is optional; thus an implementation may elect to support only parts of the complete framework. The modules are as follows.

Core The core of the framework must be provided in all implementations. The fundamental interface requirements, from the perspective of the application are:

- contract support: to be able to access a minimum, guaranteed, periodic budget;
- contract re-negotiation: to be able to request additional resources on demand;
- acceptance testing: a contract may be refused;
- to be able to query the existing resource usage.

The core provides a basic interface that allows real-time applications to be scheduled. A minimal system, for example a small embedded system, with only one or two applications may only need the core module.

Hierarchical scheduling This module provides a two layer scheduling architecture where application specific, user-selectable schedulers can be linked with a service contract. This module builds on the core module and provides a useful framework for integration of legacy applications.

Synchronization The synchronization module extends the core contract interface to allow applications to specify the critical sections between servers as part of the contracts. The module provides support for enforcing this synchronisation in both the acceptance testing and scheduling.

Spare Capacity Sharing This module allows any CPU bandwidth that is unrequested (not asked for in the basic contract) to be allocated to applications that optionally request it, above the guaranteed minimum amount. The "quality and importance" metrics are added to the contract, as well as an indication of maximum useful bandwidth.

Dynamic Reclaiming The reclamation module allows the scheduler to dynamically, at run-time, reclaim any budget that is unused in any servers, and distribute it to other servers.

Run-time and implementational specific enhancements Any implementation specific enhancements, (such as preemption levels) can be added in this generic module. This allows some room for future expansion of the framework, to incorporate novel scheduling requirements.

3.1 Contract Specification

The entire contract is given in the Table 3.1. The following sections describe the parameters of the contract in more detail.

3.2 CORE Module Parameters

The CORE module provides the basic functionality required to implement an application on FSF. The essence of the CORE is to be able to provide a minimum guaranteed service, based on a contract, to each application.

The first three parameters $C_{i,min}$, $T_{i,max}$ and D_i allow the application to specify the minimum CPU bandwidth that it requires to run correctly. The parameters reflect the server implementation of budget control. An accepted contract implies that the scheduler can guarantee $C_{i,min}$ units of CPU time every $T_{i,max}$ before D_i time has elapsed.

If D_i is omitted, the deadline for providing $C_{i,min}$ is considered to be equal to $T_{i,max}$. That is, each job may finish up until the next begins—jobs may not overlap.

The *workload* parameter specifies the nature of the work that the application is expected to do. A *bounded* workload means that each invocation of the server budget corresponds to a new release of a job. This model of computation ties the job to the underlying server mechanism, allowing finer control of the behaviour. At the end of each job, the application should declare to the system that it has finished its execution.

An *unbounded* model relaxes the synchronisation between server and jobs, allowing a single job to execute over a number of server cycles. In this case, the server acts as only a budgeting resource, allowing an application to simply execute, using a minimum bandwidth. This is very much like traditional UNIX scheduling, where the operating system has control of the bandwidth used by the application, except that in FSF, the minimum bandwidth that the application receives is guaranteed.

Deadline and Budget overruns can be detected (on bounded workloads) and signalled to the application if desired, using POSIX signals or Ada exceptions.

3.3 HIERARCHICAL Module Parameters

The hierarchical module allows each server to support a different local scheduling policy using a two-level hierarchy. The local scheduling policies that are supported are Earliest Deadline First (EDF), Fixed Priority (FP, POSIX) and none (NONE). The NONE policy implies that there is only one single thread on the server.

CORE

Parameter	Values	Observation
Minimum Budget	$C_{i,min}$	mandatory
Maximum Period	$T_{i,max}$	mandatory
Deadline	D_i	optional
Workload	Bounded/Indeterminate	default=indeterminate
Deadline Miss Notification	none/signal	default=none
Budget Overrun Notification	none/signal	default=none

HIERARCHICAL

Additional Parameter	Values	Observation
Scheduling Policy	EDF/FP/NONE	default=FP

SYNCHRONIZATION

Additional Parameter	Values	Observation
Critical Sections	$S = \{(r_0, o_0), \cdots, (r_n, o_m)\}$	optional
Provided by the Shared Objects	$C = \{C_{r_0,o_0}, \cdots, C_{r_n,o_m}\}$	optional
Preemption level	P_i	optional

SPARE CAPACITY

Additional Parameter	Values	Observation
Useful Budget	$C_{i,max}$	mandatory
Useful Period	$T_{i,min}$	mandatory
Granularity	Continuous/Discrete	default=continuous
Utilisation set	$C = \{(C_{i,1}, T_{i,1}), \cdots, (C_{i,n}, T_{i,n})\}$	optional
Quality and Importance	(Q_i, I_i)	mandatory
Signal contract changes	S	boolean; default=false

DYNAMIC RECLAIMING

Additional Parameter	Values	Observation
Useful Budget	$C_{i,max}$	mandatory
Useful Period	$T_{i,min}$	mandatory
Quality and Importance	(Q_i, I_i)	optional

RUN-TIME/IMPLEMENTATION

Additional Parameter	Values	Observation
Alternative synchronization schemes		optional

Table 3.1: Contract Parameters by Module

3.4 SYNCHRONIZATION Module Parameters

The synchronisation module provides a robust mechanism for communication and synchronization between threads running on different servers.

The mechanism is based on the notion of *shared objects* (Protected objects in Ada, similar to Hoare's Monitors in C) which have well-defined synchronization semantics.

Each shared object consists of data and an interface to that data presented as as a set of *operations*. The operations are syntactically similar to procedure calls.

However, the semantics of the operations are such that mutual exclusion is ensured between operations on the same shared object. It is implementation defined whether mutual exclusion is provided by an actual lock, or by the scheduling policy itself using a ceiling priority protocol or preemption levels.

Each operation has a stated worst case execution time associated with it, which is used to determine the blocking time that waiting for this resource can impose. The specification for each shared object (r_i) includes for each operation (o_j) the worst case execution time C_{r_i,o_j}.

Conversely, an application declares its intent to use a shared object using the interface in the synchronization module: for each shared object it uses (r) the application states which operations (o) it may to use.

The schedulability analysis tool will calculate blocking times appropriately. Blocking times are dependent on the implementation of the shared object locking. For example, if immediate priority ceiling or preemption levels are used then this value may be the worst case execution time of the largest critical section.

Nested calls between shared objects are not permitted.

3.5 SPARE CAPACITY Module Parameters

The SPARE CAPACITY module extends the core module by allowing an application to receive a higher bandwidth than it minimally needs. This is done by additionally specifying a maximum useful budget, $C_{i,max}$, and a minimum useful period, $T_{i,min}$.

When a contract negotiation takes place in the system, the generated contract for the server can be between the minimum and maximum useful values (if it is accepted). This contract provides the minimum guaranteed bandwidth that the server will expect on future invocations.

Restrictions can be placed on the resulting contract. If the application can accept any contract parameters between the minimum and maximum values then this contract request is declared *Continuous*.

Otherwise, the contract request is declared *Discrete* and a *Utilisation set* is additionally given. This is a set of pairs, (C_i, T_i), of useful values. The agreed contract (if accepted) will be one pair from the set.

Contention for bandwidth from multiple applications is handled by each contract specifying a *quality* an *importance*. These parameters specify the way that any spare bandwidth in the system is shared. They are treated as follows.

Importance is a value from the set $\{1, \cdots 5\}$ where 5 is the most important. A contract that is of higher importance than another will be allocated bandwidth in preference to the lower importance contract. That is, a more important contract can be allocated *all* available resources if requested.

Importance levels form groups within which the *quality* parameter shares resources fairly between contracts of equal importance. Quality is an integer value which represents the amount of spare bandwidth to be allocated within an importance level.

It is implicit that applications are cooperative, rather than competing for spare bandwidth. If for example three applications should receive an equal share of spare bandwidth then they should set equal values for quality. If an application wishes to request 3/4 of the *real* total spare capicity, it should request 3 times[1] the existing quality total (which may be read by any application).

If at any time, the system wishes to change a contract that previously was negotiated to be higher than the original minimum service specified in the CORE module, then the application can refuse this request if it specified the *Signal contract changes* boolean parameter.

3.6 DYNAMIC RECLAIMING Module Parameters

If a particular server has unused bandwidth at either the end of the server-cycle or when the application running on the server indicates that it does not require any further bandwidth from the server in this server-cycle then this bandwidth can be used 'reclaimed'.

This spare bandwidth is allocated directly to servers that can accept it (up to their maximum useful service, $(C_{i,max}, T_{i,min})$.

The *quality* and *importance* parameters may be used for this, however an implementation is not required to do so. It is implementation defined what scheme is used to provide this service. Capacity Sharing is one useful way; a new scheme called *Rewriting History* under development in FIRST may also be used.

3.7 RUN-TIME/IMPLEMENTATION Module Parameters

An implementation is free to specify further parameters to the contract as required. One useful mechanism that can be identified is parameters for alternative synchronisation schemes.

3.8 Portable Semantics

FSF presents an application interface that promotes portability (an application should be able to run on any FSF implementation); however the modular approach, where modules are optional and implementation dependent can cause conflicts where an application depends on a module that is not implemented.

[1] Since it wishes to have 3/4 of the total, the existing appliactions must have 1/4; therefore it requires 3 times what they aready have.

The behaviour of the system in the case that an application attempts to use an unimplemented module is defined as follows.

Runtime errors and warnings are issued using signals in the C interface and exceptions in the Ada interface. Pre-runtime errors and warnings are issued by the compiler tools or implementation defined tools as appropriate.

The CORE module is always present; omission of this module is not possible.

If the SYNCHRONIZATION module is not present in an implementation, and an application requires synchronised communication then this is a program error (permanent failure). The error can be detected pre-runtime (in likely implementations: during linking).

If the HIERARCHICAL module is not present in an implementation, and an application requires a local scheduling policy, then this is a program error (permanent failure). The error is detected at runtime. The exception to this is if the application requests policy *none* (*i.e.* there is only one process in the application); this does not an error condition.

If the SPARE CAPACITY module is not present in an implementation, and an application requests parameters provided by the SPARE CAPACITY module, this is not an error. The program should continue to run with the minimum budget as defined by the CORE module. However, a runtime or pre-runtime warning should be issued.

If the DYNAMIC RECLAIMING module is not present in an implementation, and an application requests parameters provided by the DYNAMIC RECLAIMING module, this is not an error. The program should continue to run with the budget as defined by the CORE (and optionally SPARE CAPACITY) module(s). However, a runtime or pre-runtime warning may be issued.

If an application attempts to use parameters from the RUN-TIME/IMPLEMENTATION module, the behaviour is implementation defined. Usually, this should be a pre-runtime error.

Section 4

Initialisation and Group Mode Changes

This section describes initialisation, arranging first contracts and how contracts can be considered as an atomic group.

4.1 Initialisation and New Contracts

Initialisation of an application in FIRST requires special consideration. The FSF supports dynamic and open systems, where applications may join at any time. It is apparent that a joining application should not affect existing running applications in any way that would prevent them from receiving the service as defined in existing contracts. Thus, when a contract negotiation is done for a new application, some time (a budget) is required in order to actually perform the negotiation, including acceptance tests.

The initialisation process is as follows.

- At boot-up, before the FSF scheduling architecture is running, the main 'system' program (here called *main*) initialises hardware and any initialisation required for FSF. Note that FSF scheduling is still not running.
- The main program requests a contract for each application that is to be running in the system when the system starts. The main program may also request a contract (budget) for itself.
- When all contract negotiations are complete, the main program signals the FSF scheduler to begin. The main program may then exit.

If the main program created a contract for itself then a new application may use the time assigned to that budget in order to negotiate a new contract. Therefore, the 'main' contract parameters determine the responsiveness and ability of the system to accept contract requests.

If the main program did not create a contract for itself, then the system is closed, no new applications may ask for a contract, except when they are started by another application and that original application negotiates the contract for the new application.

4.2 Atomic Contract Groups

Where an application is split over a number of different servers (hence contracts), it is important that either all of the contracts are accepted, or none are accepted—the situation where an application receives only some of its contracts should not occur.

The FIRST framework allows a set of contracts to be considered in a group. The whole group is negotiated as an atomic contract in one operation. The overall structure of a group negotiation, as viewed from the initialisation code of an application, is:

```
contract_begin_new_group(g);
contract_add_to_group(g, contract);
contract_add_to_group(g, contract);
...
contract_ok = contract_negotiate_group(g);
if (contract_ok) {
  ..
}
```

The application then has the opportunity to consider alternative contracts for acceptance.

The concept of atomic contracts extends in a distributed environment to the negotiation of contracts for the utilization of several processing resources (processors and networks) in just one negotiation. Thus complete transactions can be accommodated in the system or rejected as a whole.

Section 5

Distribution

An FSF contract may also be applied to a network to 'reserve' bandwidth on the network in a similar way to arranging a contract for CPU bandwidth. Not all of the contract is relevant for networks; only the CORE and SPARE CAPACITY modules may be provided in an implementation.

The network contract is identical to the CPU contract, except that the units of the parameters are given in bytes rather than time units.

The the network protocol is not restricted within FIRST, many network protocols may be used, for example ethernet and CAN. Implementations for both have been produced.

5.1 Notes on Distributed Transactions

An important consideration is a distributed transaction, where, for example, a process on one CPU (with its contract) sends data over the network (with another contract) to a process on another CPU (which has another contract). It is clear that the three (or more) contracts must all be compatible with each other, *i.e.* they all operate in the same mode and with the same period.

The contract negotiation for the contracts in a transaction therefore places requirements on the acceptance of a contracts, in a similar way to the atomic group of contacts (Section 4.2). Although FIRST does not directly consider the implementation of such a scheme, it is important that FIRST does support a middleware implementation.

The mechanism for this will be defined in a phase 3 document.