Mapping a Multi-Level Scheduling Pattern Language to Distributed Real-Time Embedded Applications[1]

Christopher Gill
cdgill@cse.wustl.edu
Department of CSE
Washington University
St. Louis, MO

Douglas Niehaus
niehaus@ittc.ku.edu
Department of EECS
University of Kansas
Lawrence, KS

Lisa DiPippo and Victor Fay Wolfe
{dipippo,wolfe}@cs.uri.edu
Department of CS
University of Rhode Island
Providence, RI

Lonnie Welch
welch@ohio.edu
School of EECS
Ohio University
Athens, OH

Abstract

Mission-critical Distributed Real-Time and Embedded (DRE) systems pose significant resource management challenges at and across all architectural levels, *i.e.*, the operating system and low-level middleware on each endsystem, and distributed services spanning multiple endsystems. Furthermore, the challenges posed by one application may differ from the challenges posed by another. As developers of complex DRE applications move increasingly from building individual systems to composing systems of systems, it is imperative to identify approaches that can reconcile design forces throughout a multiplicity of architectural levels and application scenarios. This paper makes two contributions to the design of resource management for DRE systems. First, it describes our recent refinements to a pattern language for resource scheduling in DRE systems. Second, it examines how the pattern language applies to several example DRE systems, thus giving guidance to developers of both individual DRE systems and composite systems.

1 Introduction

We have identified, and are continuing to extend and refine our description of, a pattern language consisting of scheduling-related design patterns at and across multiple architectural levels. We call this pattern language "Resource Rationalizer" [GNDWS], as its primary purpose is to guide the design of scheduling architectures toward a *rational* resolution of the resource constraints at and across each architectural level, with the end-to-end requirements of complex mission-critical DRE applications. This paper is structured as follows. Section 2 describes the Resource Rationalizer pattern language, and our recent extensions and refinements to it. Section 3 describes three example DRE applications in detail, and considers the paths through the pattern language consisting of the patterns used in each application. Finally, Section 4 offers concluding remarks and describes future work on the pattern language and its applications to DRE systems.

2. The Resource Rationalizer Pattern Language

Figure 1 illustrates the Resource Rationalizer pattern language. In addition, Figure 1 reflects our recent identification of higher-level architectural roles played by the patterns in the language: *control* patterns for reasoning about resource management, *actuation* patterns for adjusting resource allocations and *sensing* patterns for gathering performance and resource data. Taken together, the control, actuation, and sensing roles at each level form an integrated control architecture for monitoring, evaluating, and adapting resource allocations and steering real-time performance end-to-end. Each pattern's background shading in Figure 1 reflects its higher-level role.
In considering these roles, we have identified several additional patterns in the language. At the distributed middleware level we have identified the *Distributed Performance Monitoring* pattern, and have also divided what we formerly called the Global Resource Allocation pattern into the distinct *End-to-End Allocation* and *Service Request Allocation* patterns. At the local middleware level, we have identified the

[1] This work was funded in part by The Boeing Company, the DARPA Quorum and ANTS programs, and ONR.

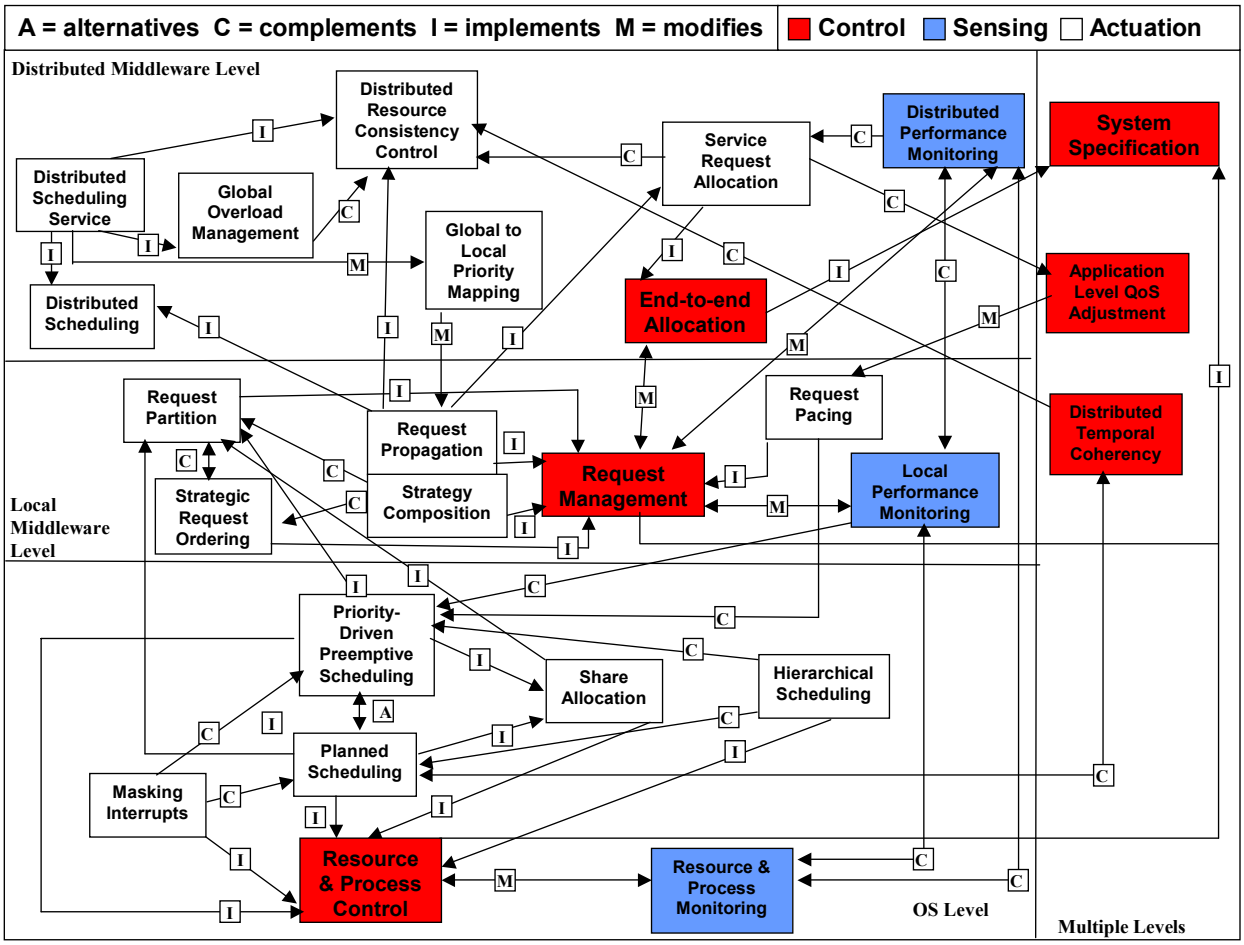

Figure 1: Resource Rationalizer Pattern Language

Request Management and *Local Performance Monitoring* patterns. At the OS level, we have newly identified the *Resource and Process Monitoring* and *Resource and Process Control* patterns.

2.1 New Distributed Middleware Patterns

Name: Distributed Performance Monitoring

Problem: When determining when and how to reallocate computing application services as well as computing and network resources, it is necessary to know the state of the distributed system. This involves the construction of accurate models of process resource needs and performance; additionally, the utilization and state of the resources must be modeled.

Context: A pool of distributed computers is interconnected via a network. A set of distributed real-time application systems are using the resources.

Forces: To provide a meaningful and consistent view of the software and resource components of a distributed real-time system requires the aggregation of monitoring information collected at lower levels. The aggregation must be performed in an efficient and accurate manner.

Solution: Define global metrics that characterize relevant aspects of performance for the software systems and important aspects of resource state.

Select and gather the monitoring information needed to calculate the metrics and compute the metrics. Be sure that frequency of information gathering is sufficient to allow the metrics to have the desired accuracy, but also be careful not to impose too large of an overhead on the system during the gathering process.

Resulting Context: Makes use of lower level metrics provided by the Local Performance Monitoring pattern, and indirectly uses metrics provided by the Resource and Process Monitoring pattern.

Rationale: Effective allocation of resources for dynamic application systems requires the ability to capture the state of the system accurately. Allocation decisions are only as good as the information used to make the decisions.

◊ ◊ ◊

Name: Service Request Allocation

Problem: When deciding among several subsystems on which to place the execution of a particular task, certain resources can become overutilized while others may be underutilized. This poor global allocation of resources could cause some tasks to unnecessarily violate timing constraints. There is a need to fit these tasks on the subsystems such that timing constraints of the current tasks are met, as well as to consider future dynamic tasks with timing constraints.

Context: A real-time distributed system in which particular tasks may use any of a set of equivalent resources from one of several operating systems or endsystems.

Forces: Choices of which endsystem to assign a task, or which resources within various endsystems to allocate to the task. Allocation needs to facilitate overall enforcement and analysis of real-time requirements. A *consistent* view of global state is needed to maintain properties such as causality. A compatible notion of scheduling policies and parameters is needed, among possibly heterogeneous application tasks. [GNDWS]

Solution: Allocate tasks to the subsystems that yield the best chance that the specified timing constraints will be met. Use schedulability techniques to predict the best subsystem on which to allocate the requested task. Further, consider future tasks when making this allocation. This may be done by examining prior distributions of tasks in similar applications, as well as by choosing subsystems on which execution time will likely be freed soon, i.e. subsystems that have aperiodic tasks ending.

Resulting Context: Load allocation techniques as described above may require some run-time analysis of current system conditions. This will incur added overhead to the execution of the application. For this reason, load allocation algorithms should be designed and implemented carefully to utilize as much precomputed system information as possible, and avoid unnecessary analysis. Alternatively, simple load allocation techniques (like first fit, or simple balancing algorithms) may be sufficient and would incur less overhead. The trade-off here is that the simpler techniques will be less predictable, potentially requiring reallocation of resources in the future.

Rationale: This pattern will allow real-time service requests to be allocated such that real-time constraints will likely be met. Further, the proactive nature of the pattern reduces the need for costly reactive reallocation of resources.

◊ ◊ ◊

Name: End-to-end Allocation

Problem: An adaptive DRE system must have the ability to decide when adaptations should occur. Additionally, it should have coherent decision logic that can (1) determine how to improve an allocation and (2) issue commands to carry out the steps needed for reallocation.

Context: Process monitoring and fault diagnosis techniques have existed for decades and have been used in the engineering of such complex systems as the space shuttle and the Aegis combat system.

Forces: The allocation of resources for DRE systems involves guaranteeing timing constraints that involve chains of multiple software components that span multiple computing and network resources. This requires knowledge and analysis of multiple resources and multiple application services.

Solution: Describe the real-time and resource requirements of the DRE application software. Define and implement a *controller* function that can determine corrective allocation needed to restore the system to its desired state. Provide an interface to the DRE application software for use by the *controller*.

Resulting Context: Applying this pattern will provide the actuation mechanisms for carrying out reallocation decisions. In addition to this pattern, applying the Resource and Process Monitoring pattern will provide the information needed for good decisions in the controller.

Rationale: This pattern helps especially in adaptive DRE systems by providing the decision-making capability that determines when and how resources are reallocated. Schedulability analysis can be applied to a proposed reallocation to determine if key real-time time constraints can be met. Techniques such as rate monotonic analysis exist for performing schedulability analysis. In a broader sense, control systems technology builds controllers that assess a plant's state and calculate specific control actions that can help to restore the plant to a desired state through control (resource allocation) actions.

2.2 New Local Middleware Patterns

Name: Request Management

Problem: Requests arriving from multiple remote endsystems, or from local endsystem events, must be managed consistently on the local endsystem.

Context: Distributed systems in which end-to-end and local timing requirements must be achieved through local enforcement mechanisms.

Forces: Queuing in low-level middleware adds overhead but is useful to reorder requests dynamically. Static priority lanes avoid overhead but require early de-multiplexing to avoid priority inversions. Multiple forms of scheduling can be applied at the local endsystem, each with its own strengths and weaknesses in the face of other forces. Furthermore, OS level support of different scheduling forms may differ across platforms.

Solution: Provide customized forms of scheduling in middleware on each endsystem to offer the most effective forms of local request management available for each operating system and application.

Resulting Context: This pattern relies on selection of a particular endsystem scheduling pattern for its implementation, such as the endsystem scheduling patterns discussed in [GNDWS].

Rationale: Supporting flexible scheduling strategies in local middleware has been shown effective for complex DRE applications[GSC].

◊ ◊ ◊

Name: Local Performance Monitoring

Problem: Some forms of scheduling in local middleware require closed loop feedback for effective management, particularly of dynamic loads.

Context: Applications where scheduling controllers at the local middleware and distributed middleware levels would benefit from more complete information about application progress and other run-time information.

Forces: Monitoring must not extract an undue penalty in overhead or even more importantly in jitter. Particular kinds of information, such as timing specifications, may cross thread or even process boundaries and must account for nuances such as locking. Correct instrumentation often requires not only that system performance metrics be taken, but details of where and under what conditions the metrics are gathered.

Solution: Use efficient techniques such as inline methods, metrics data caches, and conditionally compiled probes to instrument the local middleware infrastructure. Feed back the gathered information to dispatchers, schedulers, and higher-level resource controllers and monitors.
Resulting Context: Additional instrumentation such as application component upcall adapters for timing profiles and deadline success and failure detection are often useful at the local middleware level. In doing so, the local middleware enables metrics collection with low invasiveness to application components themselves.

Rationale: Dynamic techniques such as feedback control scheduling[LSTS], cancellation of futile operation chains[Gill], and adaptive rescheduling[Corman], rely on feedback information for performance tuning and assurance.

2.3 New OS Patterns

Name: Resource and Process Monitoring

Problem: Decision-making is important to the proper functioning of the system, and so needs to be aware of the functioning of various entities. Thus, a sensory component is needed to enable the resource allocator to make good decisions.

Context: A foundation of information sources exists in operating systems and computer hardware. Open and extendable APIs for information system services are available.

Forces: Every operating system makes such information available. Every system that tries to determine what going on in a DRE system uses such information. The scheduling function inside of an OS keeps track of such information. Exposing such information in a systematic manner is important.

Solution: Maximize accessibility to the information through a catalog (name space) of data sources. Implement functionality required for access to data sources. Select desired features from catalog and incorporate into system being constructed.

Resulting Context: Does not tie to any lower level patterns that are needed to complete this pattern.

Rationale: Good decisions require good information. Decision-making is strongly constrained by the information that is used. The information needs vary, depending on context. Having a flexible way of gathering required information is needed.

◊ ◊ ◊

Name: Resource and Process Control

Problem: The essence of the problem is the need to be able to carry out resource allocation decisions. For every resource that is to be used, access to it must be provided to allow adjustment of appropriate properties.

Context: Tool sets provided in all modern operating systems. Open APIs for accessing such tool sets.

Forces: Load balancing systems (e.g., [CONDOR], [MOSIX]) provide and employ control mechanisms such as process migration, priority control, and cache configuration. Every system that tries to dynamically control resource allocation for a DRE system needs such interfaces. Exposing such information in a systematic manner is important.

Solution: Determine the set of external control mechanisms needed in general; define APIs for accessing them. Implement functionality required to deliver the control mechanisms. Select desired mechanisms and incorporate into system being constructed.

Resulting Context: Does not tie to any lower level patterns that are needed to complete this pattern.

Rationale: This is the actuator that performs a resource allocation action at the request of the resource allocation decision maker.

2.4 New Multi-Level Patterns

Name: System Specification

Problem: All levels of the distributed system need to have access to the QoS requirements of the application and of the system.

Context: Applications in which QoS requirements must be specified in order for the proper control and adjustment to occur.

Forces: This pattern is affected by *performance constraints*, *resource allocation semantics*, *constrained resource supply*, *coordination and communication*, *activities spanning endsystems*, and *competing QoS requirements*.

Solution: Provide a consistent definition of the system QoS requirements in a form that can be enforced, e.g., by the Resource and Process Control pattern. Offer both reflective information for enforcing control law boundaries, and a priori definitions of system level limits and assurances.

Resulting Context: This pattern does not rely on other patterns to provide its functionality. It does provide system information to the Resource and Process Control pattern as well as the End-to-end Allocation pattern.

Rationale: Each level of the system can have access to the system specification and use the information as needed. Further, each level of the system can provide reflective system information to share with the other levels for future decision-making.

3 Example DRE Applications

Mission-critical distributed real-time and embedded (DRE) systems come in many forms, each with its own sets of requirements and resources. In this section we consider three separate applications, each originating from a different defense-related research program. We first give an overview of each application.

Unmanned Air Vehicle: The UAV application is a prototype system developed by BBN as part of a US Navy program at NSWC [KRKPS]. A UAV (unmanned air vehicle) is a remote controlled aircraft that provides video feeds of an engagement to viewers onboard ships and/or on the ground. Several UAVs may be active at any given time, sending video of various targets or enemy locations.

The UAV application is made up of four main parts: the video sender, the video distributor, the video receiver and the video viewer. The video sender is the process that takes the video feed from the UAV camera and sends it, through video streams, to the distributor. The video distributor receives a video stream from the sender, and sends it to video viewers on ship or on ground. The video receiver receives the video stream from the distributor and the viewer displays the video for users to analyze.

The above components make up the core of the UAV application, however other parts can and have been added to enhance the utility of the system. For example, the distributor can also send video to an automatic target recognition (ATR) system, which examines the images and recognizes key targets. From this system there might be feedback to the UAV to direct it towards the target. The UAV application can run under various scenarios, with different combinations of the above components. In the simplest case, there would be one sender, one distributor and one receiver / viewer. However, there can be multiple UAVs flying over different regions and sending video streams to multiple distributors. The distributors can send the video streams to multiple viewers, as well as other analysis systems, like the ATR.

Avionics Mission Computing: The Bold Stoke platform is a domain-specific middleware framework developed for Avionics Mission Computing applications in production military aircraft by the Boeing Company. A basic Bold Stroke application consists of avionics software components performing operational flight program (OFP) tasks, e.g., for navigation, heads-up displays, and vehicle airframe monitoring, hosted on COTS middleware and real-time operating systems and running on several mission computers connected by a VME bus or other highly efficient and predictable

interconnects. Application data is passed between processors by a domain-specific data replication middleware service, and processing and data concurrency is mediated by a real-time CORBA Event Service.

As in the UAV example, a basic Bold Stroke application may be supplemented by additional processing, filtering, and coordination components. For example, in the Weapon System Open Architecture (WSOA) program, the ability to download target imagery from a remote C2 aircraft was added to a F-15 cockpit application [Corman]. In the Adaptive Software Test Demonstration (ASTD) phase 2 program, a reasoning application was combined with the basic OFP[GSGH]. With each additional capability comes an increasingly rich set of application requirements and resource constraints, which must be resolved within the overall resource management design.

Autonomous Agent Teams: The driving application for the Autonomous Negotiating Teams (ANTS) project at the University of Kansas provides good examples of a number of the patterns within the Resource Rationalizer pattern language. The driving application consisted of a number of small radars capable of providing a range of information to the agents controlling each radar. Negotiation among agents was used to resolve conflicts over how resources were used to track targets moving through the area covered by a set of agent-controlled radars. Several agents were co-resident on a given computing platform, and thus had to negotiate with each other about access to computational resources as well as about what information would be collected from a given radar and when it would be collected.

For the purposes of this discussion the essential aspects of the ANTS application are that 1) agents were required to make decisions under time constraints, 2) more than one pair-wise negotiation between a given agent and others could proceed concurrently, 3) agents shared computing resources and thus could affect each other's execution behavior, and 4) agents had to be reflective about their own and other agents' execution behavior because they had to be able to negotiate about computing resources as well as information from specific radars.

The computational demands on the system varied from fairly static to quite dynamic, depending on the nature of the tracking task. Significant execution mode changes occurred when a target was first detected, since resources were allocated to tracking it, which required the agent noticing its existence to start negotiating with agents controlling radars that had the potential to track it as it moved in any of several possible directions. The agent support structure, including both middleware and operating system components, made a wide range of information available to thread within an agent, and provided a number of ways in which an agent could control its own execution.

3.1 Unmanned Air Vehicle Scheduling Requirements

The main real-time requirement of the UAV application is that video streams must be delivered in such as way as to provide a display that is easily viewable by a human (or by an application like the automatic target recognition system described above). This might require, for instance, that the frames of an MPEG video be delivered at a rate no less than 30 frames per second.

Further real-time requirements are added to the application when systems such as ATR are included. In this case, control signals sent to the UAV must be delivered in a timely fashion so that the UAV can react and move towards the required target before it moves out of range.

To meet the real-time constraints imposed by the above requirements, the UAV application may need to adapt to dynamic system conditions. For example, if a distributor host is receiving and sending a heavy load of video, it may become overloaded. In this case, some of the load may need to be allocated to a distributor on a different host. Also, when the ATR system finds a critical target, it is crucial that the control signal be delivered to the UAV on time. If there is heavy load on the ATR host, or on the UAV itself, this time constraint may be violated. Thus, the importance or criticality of the control task, and all other tasks in the system, must be taken into account when determining what load may be sacrificed in order to maintain overall system timeliness.

To demonstrate the usefulness of the Resource Rationalizer pattern language, we trace through the language here to show what parts are necessary to support the real-time requirements of the UAV application. We will look at two different versions of the UAV application: a static system and a dynamic system.

3.1.1 Static UAV Application

The static UAV application consists of a single UAV, one distributor and one receiver / viewer pair. Each of these components resides on a separate host. The system remains static in that no components are added dynamically, and the real-time requirements, such as execution time, periods, deadlines, are all well-known, and don't change. On each host, the video stream can be represented as a periodic task with timing constraints imposed by the required frame rate.

We present this simple application for several reasons. First, it indicates that the pattern language can support this type of static, real-time system. And second, it provides a contrast to the dynamic application that we describe next. This allows us to demonstrate the wide range of types of systems that the pattern language can support.

Distributed Middleware Patterns: On the distributed level, the static UAV application applies the Distributed Scheduling Service pattern to provide global scheduling of the various periodic tasks across the system. The Distributed Scheduling Service pattern implements the Distributed Scheduling pattern, which provides global scheduling parameters, such as priority assignments, to the tasks. The Global to Local Priority Mapping pattern is necessary to provide a mapping from global priorities to local priorities, if priority based scheduling is being used.

The static UAV application also requires the Service Request Allocation pattern, as implemented by the End-to-End Allocation pattern. These patterns would be used once, at system set-up time. Once the allocations are made, they will not change.

Local Middleware Patterns: On the local middleware level, the static UAV application requires the Request Management pattern to map service requests onto local resources and processes. In particular, the Request Propagation pattern is necessary to map the requests onto resources of the various endsystems. This pattern also implements the Distributed Scheduling pattern in the Distributed Middleware level.

Operating System Patterns: The UAV application requires the Resource & Process Control pattern. Depending upon the kind of scheduling that is used in the UAV application, it will require the Resource & Process Control pattern to implement either the Planned Scheduling pattern or the Priority-Driven Scheduling pattern. The decision of which type of scheduling to use will affect all of the levels of scheduling patterns.

Multi-level Patterns: In the static UAV application, the only multi-level pattern that is required is the Distributed Temporal Coherency pattern. It is critical that the system maintain a consistent view of time so that the required timing constraints can be understood and upheld on all levels of the hierarchy, as well as all parts of the distributed system.

3.1.2 Dynamic UAV Application

The dynamic UAV application consists of several UAVs, flying in different regions, which may intermittently send video streams when they reach a particular location. There are several distributors that can receive the video streams, and these distributors can send the video streams to various receivers that will display the video on a viewer, or will provide input to an ATR system. The dynamic nature of this application makes it impossible to predict which tasks will execute when, and therefore will rely on dynamic scheduling, and load management. The specific tasks in this application that need to be schedule include the periodic execution of video sending and receiving on the UAV hosts, the distributor hosts, and the receiver / view hosts. Also, tasks within the ATR that recognize and respond to targets found in the video streams must be scheduled as well. These tasks will be aperiodic, as they will occur dynamically when targets are found. In this section we describe the patterns in the Resource Rationalizer pattern language that this dynamic UAV application would use.

Distributed Middleware Patterns: As in the static version of the application, the dynamic UAV requires the Distributed Scheduling Service pattern and the Distributed Scheduling pattern. In this case, the specific algorithms that will be used to implement the strategies within the patterns must be able to assign real-time parameters (such as priorities) to both periodic (video streams) and aperiodic (ATR) tasks. The Global to Local Priority Mapping pattern is again necessary if the application is implemented using priority-based scheduling.

The Service Request Allocation pattern is required in the dynamic UAV application to allocate the tasks to resources. As in the static UAV, this pattern will implement the End-to-End Allocation pattern that is required to ensure that the allocation of tasks across the system will meet the specified timing constraints. Specifically, this may involve choosing one distributor process over another to handle a particular video stream based upon the current and expected loads on the hosts involved. Unlike the static UAV, this dynamic application may also require reallocation of resources if any of the hosts becomes, or is predicted to become, overloaded. At this level, the Global Overload Management pattern can be used to determine how to handle the problem. For example, if the host containing the ATR becomes overloaded, and it has found a critical target, the ATR's task(s) may be deemed more important, or critical, than other tasks on that host. Those less important tasks may need to be sacrificed, or reduced in quality in order to continue to meet the timing constraints of the important ATR tasks.

Local Middleware Patterns: At the local middleware level, the dynamic UAV application requires the Request Management pattern. In this case, this pattern can be implemented using the Request Propagation pattern, the Request Pacing pattern, and/or the Strategy Composition pattern. This is because in the dynamic application it may be necessary to allow the endsystem to adjust to the dynamic system conditions. For example, the Request Pacing pattern could pace the individual frames in the periodic sending process in order to keep them separated in time. The Strategic Request Reordering pattern may also be used if certain tasks are deemed to be more important than others, such as the ATR example described above.

Operating System Patterns: The dynamic UAV application requires several operating system level patterns in order to ensure that the scheduling decisions made at the higher levels are enforced on the individual hosts within the distributed system. As with the static application, the Resource & Process Control pattern can be implemented using either the Planned Scheduling pattern or the Priority-Driven Preemptive Scheduling pattern. However, the dynamic version of the application also requires the Resource & Process Monitoring pattern because the system must be able to monitor the system to determine if timing constraints are being upheld.

Multi-level Patterns: Several multi-level patterns are required in the dynamic UAV application, which are not necessary in the static version. This includes the Software Performance Monitoring & Diagnosis pattern, the System Level Control pattern, and the Application Level QoS Adjustment pattern. Each of these patterns is necessary to allow each of the levels in the pattern language to monitor the system, and to adapt to the changing system conditions. For example, if a new viewer process becomes active, and it begins to receive a video stream from a particular distributor, the host that the distributor is on could hit some load threshold that indicates that the system is almost overloaded. In this case, the Software Performance Monitoring & Diagnosis pattern will recognize the potential problem by using the OS level Resource & Process Monitoring pattern, as well as the Service Request Allocation pattern to help determine where the request should be reallocated. Then the Application QoS Adjustment pattern will determine a different distributor from which the viewer can receive the video stream. This will be done in conjunction with the Service Request Allocation pattern and/or the Request Pacing pattern. The System Performance Monitoring & Diagnosis pattern will also feed back, through the System Level Control pattern, to the Resource & Process Control pattern at the OS level.

3.1.3 Further Discussion

The description of the patterns used in this application assumed that certain scheduling decision have not been made – i.e. priority-based scheduling vs. planned scheduling. This assumption was made so that the discussion above could consider both possibilities. If one option were chosen over another, the path through the pattern language that is required by the application would be more direct.

3.2 Patterns to Support Bold Stroke Avionics Mission Computing

The real-time requirements of Bold Stroke applications center on a periodic frame-based model, in which each invocation of an OFP component is invoked and must complete within one periodic frame. Different components may run at different rates, and concurrency is enforced by a set of dispatching threads pushing events from a real-time CORBA Event Service to the components. Further real-time requirements are introduced when features such as image download management or reasoning networks are added. In this case, the additional components must be carefully scheduled so they meet their own real-time deadlines, but do not interfere with the deadlines of critical OFP components.

To meet the real-time constraints imposed by the above requirements, the Bold Stroke framework may also need to adapt to dynamic system conditions. However the range of adaptation may span invocation-to-invocation dynamic scheduling all the way to manipulating imagery compression to improve download times. Dynamic scheduling allows the framework to reorder non-critical invocations so that more of them can meet their deadlines, while still isolating critical invocations from non-critical load. Adaptive re-scheduling of tasks can be used to ensure tasks are appropriately assigned to frames where they can meet their deadlines.

We continue our examination of the Resource Rationalizer pattern language by considering the patterns necessary to support Bold Stroke applications. We describe three variations: a basic OFP, an adaptive OFP, and an OFP with autonomous reasoning.

3.2.1 Basic Operational Flight Program

As in the UAV discussion, we first present a simple version of the Bold Stroke application to show direct applicability of the pattern language, and to motivate how systems beyond those described in this paper can be supported through incremental application of additional patterns in the language.

The basic application consists of distributed avionics software components running on COTS real-time middleware, operating systems, and middleware. Specialized functions such as image display processing may receive support from dedicated hardware, but even such specialized components communicate to other components via the standard COTS infrastructure. Concurrency of component invocations is mediated by a real-time event service, within which real-time dispatching is implemented using a number of the patterns in the language.

Distributed Middleware Patterns: The *Global-to-Local Priority Mapping* pattern was applied manually in the early implementations of Bold Stroke to implement a form of end-to-end priority lanes. Recent migration of Bold Stroke to Real-Time CORBA 1.0[PSC] interfaces provided by TAO[2] makes this pattern even more relevant in this example. A loose form of the *Distributed Scheduling* pattern is seen in this example as well, with local schedulers informed of local method invocation rates by components on other endsystems.

Local Middleware Patterns: The *Strategic Request Ordering* pattern was applied to dynamically dispatch non-critical invocations dynamically, while dispatching critical invocations statically. The *Request Partition* pattern was applied to dispatch critical invocations in different queues and by different threads at higher priorities, than non-critical invocations. The Strategy

[2] TAO is an open-source real-time ORB available at http://www.cs.wustl.edu/~schmidt/TAO.html

Composition pattern was applied to combine static and dynamic scheduling strategies, with appropriate preservation of criticality isolation.

Operating System Patterns: The application depended on a POSIX-like COTS operating system, with *Priority-Driven Preemptive Scheduling* as the primary OS-level pattern used by the application.

Multi-level Patterns: The *System Specification* pattern was applied at all levels of the application, from the hardware to the operating system to the middleware, to the distribution and components and their specifications.

3.2.2 Adaptive Operational Flight Program

The adaptive OFP application consists of an F-15 cockpit communicating with a remote C2 aircraft to download imagery for enroute mission redirection. The F-15 cockpit system is fundamentally an extension of the basic OFP application to include dynamic adaptation of QoS for downloaded image tiles. The main trade-off is between image quality and download timeliness, with compression levels used to control this trade-off. In addition to the patterns described in the basic OFP application example, the following additional patterns were applied to the adaptive version.

Local Middleware Patterns: The *Local Performance Monitoring* pattern was applied to provide feedback to higher-level resource management by QuO[ZBS] and a Real-Time Adaptive Resource Manager (RTARM) [HJHMLKSZB]. In addition, the *Request Management* control pattern was applied to reschedule not only priorities of requests, but also rates of execution of components upon direction by the RTARM.

Multi-level Patterns: The Application Level QoS Adjustment pattern was applied to define and allow reasonable variations in image tile quality.

3.2.3 Operational Flight Program with Autonomous Reasoning

The OFP application with on-line reasoning consists of an advanced reasoning-based avionics application[MJ] running a subset of the OFP components used in the basic OFP example. While OFP component invocations are engineered to be highly predictable, reasoning tasks are inherently less bounded, and therefore must be designed carefully to mesh with the periodic frame-based scheduling model used by the OFP. In addition to the patterns described in the basic and adaptive OFP application examples, the autonomous reasoning OFP example applies the following patterns:

Local Middleware Patterns: The *Strategy Composition* pattern is applied in a new way in the autonomous reasoning example – not only are static and dynamic dispatching combined for OFP components, but task network component dispatches are mapped into the OFP dispatching infrastructure, so that the rates and priorities at which reasoning tasks are dispatched are reconciled with those of OFP tasks. In addition, the *Request Propagation* pattern was applied to tunnel dispatches from the OFP dispatcher to a higher level dispatcher in the task network itself.

3.2.4 Further Discussion

In general, the progression from the basic OFP to the adaptive OFP to the reasoning OFP showed a cumulative application of patterns in the language. Additional patterns, the majority applied at the local middleware level, are needed to resolve the new design forces introduced by each evolution of the OFP example. One observation worth noting is that the example assumes a POSIX OS, which means the Priority-Driven Preemptive Scheduling pattern is necessarily the basis for implementing the Request Partition pattern at the local middleware level.

3.3 Patterns to Support Teams of Autonomous Agents

Real-time execution constraints arise in a number of different ways in the ANTS application. The most obvious is that as a target moves through the tracking area, decisions about which radars will measure its progress must be made while those measurements are still possible. Negotiation is required among agents to resolve any conflicting demands on a given radar, which can arise not only because it is being asked to watch more than one target during the same time period, but also because it is currently out of service due to calibration, or because the agent controlling it is reluctant to consume the power required to perform the requested service. The negotiations themselves have deadlines deriving ultimately from the requirements of the tracking task, as a given agent must ensure that it can resolve a given negotiation in time to try another if the current negotiation fails.

This application area is thus an unusual venue for real-time in that agents are plagued by incomplete knowledge of the situation, conduct computations that have no worst case execution time, at least no known worst case, and yet they must try to satisfy deadlines, using their awareness of various aspects of the situation, and the control over the use of system resources provided to them. While agents cannot guarantee to succeed, they are nonetheless responsible for doing as well as they can, which requires both an awareness of the system state, the overall goals of the system, but also a willingness to cooperate to achieve the best system-wide result.

One of the most interesting features is the dynamic nature of both the system requirements, and the resource allocations designed to try and satisfy the requirements. The agents' effort to adapt their behavior to circumstance was supported by the real-time scheduling server, the real-time scheduling services library, and the OS facilities used to allocate resources and to monitor their use.

3.3.1 Real-Time Scheduling Server

This portion of the ANTS application consisted of a server with which each thread within a multi-threaded agent negotiated to obtain CPU resources, which was expressed as a percentage of CPU during a given time period. The scheduling server was a central point at which all such requests from agents on a given computer were resolved, and a new explicit execution plan satisfying the selected agent thread execution constraints was constructed. Note that not all requests were satisfied, as the total requested CPU could easily exceed the total available.

Feedback to the agent threads could include counter-offers to the agents' requests, as well as simple information about what other agent threads were possible candidates for negotiation, since they were current holders of the desired resource. Agents could then revise their requests to conform to current conditions, or they could go gather further information and then negotiate with current consumers of the CPU to obtain more.

Distributed Middleware Patterns: The scheduling server exhibited no distributed patterns explicitly, although it was responsible for handling overload in the form of excessive resource requests. This is an indication that it may play a role in supporting the *Global Overload Management* pattern.

Local Middleware Patterns: The scheduling server was an implementation of the *Request Management* pattern, and also the *Request Partitioning* pattern. The way in which the scheduling server constructed the agent execution schedules also included some aspects of the *Request Pacing* pattern, since the constructed schedules also controlled the rate at which the requested CPU resources were delivered to the threads.

Operating System Patterns: At the operating system level, the scheduling server took advantage of the *Planned Scheduling* pattern within the KURT-Linux operating system to implement the *Share Based Scheduling* pattern. This approach used the *Resource and Process Control* pattern within the kernel to accomplish its goals.

Multi-level Patterns: The scheduling service implementation spanned the local middleware and OS levels of the system. Schedule construction took place at the middleware level, but the execution of the constructed execution plan necessarily took place within the OS.

3.3.2 RTSS Library

The Real-Time System Services library provided an API to the RTSS for use by the agents implementing the simple protocol used by agents to request CPU resources. It also provided other services.

The CPU consumption behavior and current allocation for each agent thread was measured by the OS and made available to all threads through an RTSS library interface to new OS capabilities. Agent threads when looking for threads currently using significantly less than their allocated CPU share used this information, so they might be able to negotiate a transfer. The RTSS library also implemented an interface that permitted agents to adapt their behavior to the passage of time. Most actions of an agent had a deadline by which it should be completed. The foundation of the approach was the use of a simple set of state variable values to express the passage of time relative to the deadline. This system thus made it fairly simple to have the agent code follow one path through its code when it was still "early" in its execution period, but to decide to follow a different and more expedient path when it was getting "late" as the deadline became "close".

Local Middleware Patterns: At the middleware level, the RTSS implements the *Request Partitioning* pattern since it has to resolve conflicts among a set of potentially infeasible resource requests, deciding to actually perform some feasible subset. The *Request Propagation* pattern is also supported in the RTSS, albeit somewhat weakly, when one agent begins negotiation with another, as the deadline of the negotiation now influences what happens on the systems supporting both agents. The *Strategic Request Reordering* pattern is implemented in the negotiation between the agents, as supported by the RTSS library, and the scheduling server.

Operating System Patterns: The *Share Allocation* pattern is implemented by the scheduling server and library, which build an explicit execution plan using the explicit plan pattern to implement the shares. *Hierarchical scheduling* was strongly motivated by this application as we wanted to be able to express the idea that a given set of resources was allocated to the set of threads performing a negotiation, two threads in different agents, or that we wanted to allocate resources to the set of threads implementing the agent as a group. The current model only permitted allocation to threads individually. The best example was that a "negotiation" thread was actually a *pair* of threads, which were effectively co-routines. Only one executed at a time, while the other was blocked. These pairs should at least have been given CPU resources as a group. *Synchronous locks* were used between these pairs.

Multi-level Patterns:
The RTSS library implementation of the subsystem making agent thread resource use status available to other threads was a multi-level combination of the *Local Performance Monitoring* pattern at the local middleware level, and of the *resource and Process monitoring* pattern at the OS level.

3.3.3 Data Streams

The data streams approach to performance data gathering was used to make both system level and user level performance information available for system evaluation, and in some cases for use by agents during system execution.

Distributed Middleware Patterns: Distributed scheduling is present in the sense that an agent is negotiating into the future about which agents will track a target, and what agents will assume responsibility for further tracking as a target moves farther and farther away from its original observer. In this case it was implemented at the application level.

Local Middleware Patterns: The Data Streams User Interface (DSUI) implements the *Local Performance Monitoring* pattern.

Operating System Patterns: The Data Streams Kernel Interface (DSKI) implements the *Resource and Process Monitoring* pattern.

Multi-level Patterns: We used the Network Time Protocol (NTP) tools to implement the *Distributed Temporal Consistency* pattern to help keep performance measurement and agent based control

of system behavior coordinated across computer system boundaries. The enabled agents to talk about an absolute time at which to perform actions, and then be able to do them simultaneously, within the limits of the clock synchronization.

4 Conclusions and Future Work

In this section we summarize the contributions of this paper, in terms of concrete recommendations to developers, based on our observations on both the Resource Rationalizer pattern language and the example systems we've studied. We also note areas of future work related to this effort, particularly in the area of combining features of the systems studied, and the corresponding mapping and combination of the segments of the pattern language that apply to each.

4.1 Recommendations

Based on our experiences mapping the Resource Rationalizer pattern language to several DRE systems, we have several recommendations for developers seeking to apply large-scale pattern languages.

Identify related systems and map the language to them as well. As each system is developed within its own set of design forces, the particular patterns in the language that apply may differ from system to system. Each such mapping reveals new insights into the paths through the pattern language that are most common, and therefore presumably most important. Furthermore, differences between the paths may serve to distinguish key classes of applications.

Look for higher-level relationships among the patterns. Common paths through a large-scale pattern language are likely indicators of underlying structure. When different patterns play similar roles, either for different problems or in different contexts, it may be useful to look for deeper structure. For example in the Resource Rationalizer pattern language, an overarching controller-actuator-sensor architectural structure is evident.

Consider whether new patterns are revealed. When considering both the mapping to particular examples and the higher-level structure of a pattern language, it may be useful to examine a kind of *closure* on the language: whether new patterns are revealed by these examinations.

4.2 Future Work: System Composition

We conclude by noting that as developers increasingly move from building individual systems to composing systems of systems, the paths through pattern language that each individual system traces may serve as a guide to their integration. We describe two forms of integration, and as a "thought experiment" explore ways in which the applications described in this paper might notionally be combined, based on the patterns they embody.

System of Systems: The first kind of integration involves a system-of-systems approach in which individual systems are interconnected though possibly loosely coupled to form a larger system. For example one can envision a military battlespace infosphere[Cybenko] in which fighter aircraft, unmanned aerial vehicles, and autonomous ground robots all communicate and coordinate their activities through a central C2 aircraft. This kind of coordination requires specification of additional end-to-end real-time parameters that are mapped into each subsystem, and possibly an overarching control function provided by the C2 system. We believe that even for loose coordination of sufficiently complex systems, additional research is needed into the best techniques for applying scheduling patterns broadly at all levels of the Resource Rationalizer language across the combined system.

Composite System: Issues of control loop tightness and pervasiveness of integration of the individual systems define a spectrum from loosely to tightly integrated systems. The notion of a tightly integrated composite system lies at the opposite end of the spectrum from the loosely federated system of systems approach described above. For example, we might consider extensions to the reasoning OFP described in Section 3.3, which combine the elements of the ANTS and dynamic UAV applications. Closely integrating dynamic video

streaming, a reasoning OFP, and negotiation subsystems within each UAV would support collaborating *teams* of UAVs with increased autonomy and robustness in target acquisition and tracking, particularly for elusive time-critical targets.

To achieve the fidelity and pervasiveness of control needed in this much closer degree of integration, we believe new research is needed to examine the fine-grain interactions between the paths through the language for each subsystem, particularly where some design re-factoring is indicated by the new combinations of design forces. Furthermore, new inter-level interactions within the pattern language itself appear to be important areas of research. For example, it is useful to consider how share-based and priority-based scheduling patterns can be inter-related for appropriate isolation of critical and non-critical functionality.

We conclude by noting a few research questions that span the spectrum of loose to tight integration of scheduling throughout a combined system. Does the choice of composing systems of systems versus integrating more closely depend largely on issues of scale? *I.e.*, beyond a certain point, does close integration become intractable, mandating a system-of-systems approach? If so, at what point does this transition occur? Does that observation in turn offer insights into how to push the transition farther out, allowing higher fidelity integration of increasingly large and complex systems of systems?

References

[CONDOR] M. Lizkow, M. Livney, and M. Mutka, *Condor: A hunter of idle workstations*, Proc. 8th International Conference on Distributed Computing Systems, San Jose, June 1998.

[Corman] D. Corman, *WSOA-Weapon Systems Open Architecture Demonstration - Using Emerging Open System Architecture Standards to Enable Innovative Techniques for Time Critical Target (TCT) Prosecution*, Proceedings of the 20th IEEE/AIAA Digital Avionics Systems Conference (DASC), Daytona Beach, FL, October, 2001.

[Cybenko] G. Cybenko, *et al.*, Final Report of the AFOSR/IF Workshop on Infospheric Science, George Mason University, July 2001.
http://actcomm.thayer.dartmouth.edu/task/InfosphereFinalReport.ppt

[Gill] C. Gill, *Flexible Scheduling in Middleware for Distributed Rate-Based Real-Time Applications*, Doctoral Dissertation, Washington University, St. Louis, MO, December 2001.

[GNDWS] C. Gill, D. Niehaus, L. DiPippo, V. F. Wolfe, V. Subramonian, *Resource Rationalizer: a Pattern Language for Multi-Scale Scheduling*, 9th Conference on Pattern Languages of Programs, Monticello, IL, September 2002.

[GSC] C. Gill, D. Schmidt, R. Cytron, *Multi-Paradigm Scheduling for Distributed Real-Time Embedded Computing*, IEEE Proceedings Special Issue on Modeling and Design of Embedded Software, October, 2002.

[GSGH] C. Gill, D. Sharp, P. Goertzen, J. Hoffert, *An Evolution of QoS Context Propagation in Event-Mediated Avionics Software Architectures*, Proceedings of the 20th IEEE/AIAA Digital Avionics Systems Conference (DASC), Daytona Beach, FL, October 2001.

[HJHMLKSZB] J. Huang, R. Jha, W. Heimerdinger, M. Muhammad, S. Lauzac, B. Kannikeswaran, K. Schwan, W. Zhao and R. Bettati, *RT-ARM: A Real-Time Adaptive Resource Management System for Distributed Mission-Critical Applications*, Workshop on Middleware for Distributed Real-Time Systems, RTSS-97, IEEE, San Francisco, CA, 1997

[KRKPS] D. Karr, C. Rodrigues, Y. Krishnamurthy, I. Pyarali, and D. Schmidt, *Application of the QuO Quality-of-Service Framework to a Distributed Video Application*, Proceedings of the 3rd International Symposium on Distributed Objects and Applications, OMG, Rome, Italy, September 2001,

[LSTS] C. Lu, J. Stankovic, G. Tao and S. Son, *Feedback Control Real-Time Scheduling: Framework, Modeling, and Algorithms*, Journal of Real-Time Systems, Special Issue on Control-Theoretical Approaches to Real-Time Computing, Kluwer, 2002.

[MJ] B. McBryan and M. Joy, May 1999, *Rotorcraft Pilot's Associate Shared Memory Task Network Architecture*, AHS International Forum 55 Proceedings, AHS.

[MOSIX] A. Barak and O. La'adan, *The MOSIX Multi-computer Operating System for High Performance Cluster Computing*, Journal of Future Generation Computer Systems, 13(4-5), March 1998, 361-372.

[PSC] I. Pyarali, D. Schmidt, and R. Cytron, *Achieving End-to-End Predictability of the TAO Real-time CORBA ORB*, 8th IEEE Real-Time Technology and Applications Symposium, San Jose, CA, September 2002.

[ZBS] J. Zinky, D. Bakken and R. Schantz, *Architectural Support for Quality of Service for CORBA Objects*, Theory and Practice of Object Systems 3(1), John Wiley and Sons, 1997.