Second Draft: 7/3/2001

Specifying and Monitoring GrADS Contracts

Ruth Aydt, Celso Mendes, Dan Reed, Fredrik Vraalsen
Pablo Research Group

1 Introduction

In this document we define a *contract* in the context of the GrADS project and describe the current software infrastructure for creating and monitoring contracts. The mechanisms presented are quite flexible and support a wide range of contract specification and verification implementations. This flexibility allows for experimentation to identify the types of contracts that are most effective for applications in the Grid environment targeted by the GrADS project.

2 The GrADS Contract

A GrADS *contract* states that provided

- given *resources* (computational, network, input/output, etc.),
- with certain *capabilities* (flop rate, expected load, clock speed, memory, bandwidth, latency, transfer rate, disk capacity, etc),
- for particular *problem parameters* (matrix size, confidence interval, image resolution, etc.),

the application will

- achieve *measurable performance* (sustain F flops per second, transfer B bytes per second, render R frames per second, finish iteration I in T seconds, etc.)

during execution.

Creating or specifying a contract in the GrADS system consists of "filling in the blanks" of the above template with specific commitments of *resources, capabilities, problem parameters,* and *measurable performance* for an application. The types of resources, capabilities, problem parameters, and measurable performance metrics that are specified can vary from contract to contract. For example, one contract may commit to processor resources with certain memory capabilities, while another contract may specify input/output resources and disk capacity and transfer rate capabilities. In practice, we expect the list of types of resources, capabilities, and measurable performance metrics actually used in contracts to be fairly concise.

Strictly speaking, a GrADS contract is *violated* if any of the resource, capability, problem parameter, or measured performance specifications in the contract do not hold during the actual execution of the application. Here we see the analogy to a legal contract, which is broken if one or more parties do not fulfill their commitments/obligations as set out in the contract document.

GrADS contract *monitoring* is conducted as the application executes to verify that the specific commitments set forth in the contract are being met. If any of the commitments are not being met, then the contract can be deemed violated. One or more processes that are not part of the application being executed perform the monitoring. The monitoring processes may or may not be running on the same resources as the application.

Optionally, the monitoring system may attempt to determine and report which of the commitments were not fulfilled. The monitoring system may also generate a record of the observed behavior for use in either real-time or post-mortem analysis and tuning of the contract itself.

In practice, it is likely that one or more of the commitments made in the contract cannot be specified with complete certainty. For example, the bandwidth capability may be expected to vary between 100 and 200 Kbytes/second, or the measurable performance commitment may be a range of values from 30 to 40 frames per second. This uncertainty can be expressed as a lower and upper bound of acceptable values, as a confidence interval around a given value, or, on the monitoring side, as a degree of tolerance for measurements falling outside the contract commitment.

In addition to uncertainty for a given commitment, it is also possible that a single monitoring measurement will fall outside the promised commitment, but that over a period of time the commitment will be met. For example, if the number of frames rendered is measured every second; the values seen could be 39, 40, 7, 40, and 35. Here the rate of 7 frames per second is below the earlier commitment, but the achieved frames per second over the five-second time interval (32.2) fulfills the commitment.

Uncertainty in commitments and variations over time, which also exist on traditional computing platforms, are exacerbated by the dynamic nature of the Grid. Therefore, it is critical for the GrADS contract monitoring system to accommodate uncertainty in one or more of the contract commitments, handle outliers in a reasonable manner, and optionally report only those contract violations that exceed some configurable level of severity.

3 Contract Components

In this section we examine the various components of the contract in more detail and discuss the implications of each for contract specification and validation.

3.1 Resources

The set of resources that the application will use is known at the time the contract is written. It is assumed that this set has been chosen in a resource selection phase, and has been confirmed to be available at the time of application launch. It is also assumed that if the application behavior depends on the placement of individual tasks on particular resources, the assignment of the tasks to the resources is somehow conveyed in the specification of the resources.

The resources constitute one set of commitments that may be specified with complete certainty. For example, a system is or is not available, a file system is or is not accessible, a network between two systems does or does not exist, etc.

System status can be monitored through queries to MDS, by issuing ping commands, or by checking NWS data. Similar techniques can be employed for the other types of resources.

3.2 Capabilities

Capabilities are closely tied to the resources as they give information about the capacity of the resources to perform useful work. It is expected that when the contract is written, resource capabilities are known with some certainty. The capabilities commitments will likely come from a combination of NWS information and MDS entries.

Monitoring capabilities commitments can be difficult. One can watch NWS reports for the resources involved throughout execution, but it is difficult to determine if a given resource is being used for the application being monitored or for another unrelated task. If the monitored capability goes to zero, that is an indication that in fact the resource has failed (see previous section). Another possible method for monitoring capabilities is to stop the application periodically and run a small benchmark suite to determine what compute, communicate, memory, and disk resources the application could presumably be using.

3.3 Problem Parameters

Many applications have a unique set of problem parameters that determine to a large part the amount and type of work that will be done. When the application for which the contract is being written has such parameters, specifying them in the contract allows the contract to be more specialized. In some cases, the execution progression will be data dependent and the problem parameters will not be known in advance.

When problem parameters have been specified in the contract, probes can be put into the program to verify that the actual parameters in fact match those that were committed to. In practice, the extraction of program parameters from application job submission is usually automated and therefore reliable, and the verification of these commitments is not of particular interest once initial problems with the extraction have been resolved.

3.4 Measurable Performance

Regardless of the application, in order for a contract to be written and monitored, there must be measurable performance commitments associated with the application.

If the application algorithm is well understood, the application developer may provide a model of the application performance that takes as input the resources, capabilities, and problem parameters, and produces a commitment of performance that can be measured. For example, if the application is a graphics renderer, given a set of processors, flop rates, image size, and resolution as input, the expected frames per second rate could be produced. This is a measurable performance commitment, which could be monitored by incrementing a counter in the application each time a frame is rendered and reading that counter every second.

For many applications, the algorithm may not be well understood or the developer may not be available to help formulate a model. In these cases it may be possible through compiler analysis to generate a model of measurable performance. For example, suppose the compiler can determine the number of floating point operations and the number of bytes read and written within a loop or subroutine. Those determinations, together with processor and file system capability and problem parameter information, can be used to predict expected flops/second and bytes/second. Probes inserted by the compiler to measure elapsed time for the loop or subroutine can measure the actual achieved rates.

In some cases the source code may not be available, or may be so complex that the compiler cannot easily generate a model. In these cases, historical data may be used to form a model of the expected application behavior and the contract measurable performance would be based on that.

It may also be possible that no historical data is available, in which case the contract may state the types of measurable performance metrics that are being committed to (i.e., flops/second, frames/second, etc.) but the range of committed values may be “enormous”. As the application executes and data is collected, the range of committed values may be narrowed, based on what has been seen and assuming that past behavior is a reasonable predictor of future behavior. (It’s not a mutual fund!)

Much work is currently being done in the GrADS project to identify useful and practical measurable performance metrics. First, the metrics chosen must be things that can be monitored. For example, a commitment to be “half way done in 3 hours” is not useful unless it is possible to determine when the application is indeed half way done. Second, the metrics must be something that can be predicted for a given set of resource, capability, and problem parameter inputs. If an application's behavior varies widely based on data-dependencies that cannot be know at startup, how can a model for this application be developed that will allow some commitments to be made and monitored?

In general, when a violation of a measurable performance commitment is detected, it could be because 1) the model used to make the commitment is faulty, or 2) the measurement is inaccurate, or 3) the "prerequisite" resource, capability, and program parameter commitments have been violated, causing the application's performance to be something other than the original prediction. Considering these in reverse order, verification of "prerequisite" commitments has been covered in the previous discussions of contract components. By relying on well-established measurement techniques and measuring at a reasonable granularity, one would expect the measurements to be accurate. Detecting faulty models is difficult, but not hopeless.

One technique for detecting problems with performance models is to monitor not only the measurable performance metrics, but also the *expected execution behaviors* upon which those performance metrics are based. For example, if the model commits to a given code segment achieving N flops per second, chances are the model derived N from an expected execution behavior that the code executes some number of floating point operations for the given program parameters and the resources have the capability of executing some number of floating point operations per second. By monitoring the actual number of floating point operations in the code segment, the contract monitoring system can verify that the number executed matches the expected execution behavior. If it does not, then the model is basing its measurable performance commitment on incorrect underlying assumptions about the number of floating point operations that will be executed. "Sanity checks" such as this can be useful in evaluating models generated by humans, compilers, historical data and evolving execution patterns. While exact matches of expected and monitored execution behaviors may be unreasonable, wide discrepancies can expose faulty model assumptions.

In the case where the underlying model assumptions about expected execution behavior are correct, but the measurable performance commitments generated by the models are frequently wrong, the formula for predicting performance is itself at fault. Capturing and periodically reviewing the committed versus the measured performance in cases where all other variables (prerequisite commitments and expected execution behaviors) are consistent with model assumptions can be helpful in identifying problems with the model's formula.

4 Contract Monitoring Philosophies and Requirements

Within the GrADS project, participants have expressed an interest in monitoring contracts in a variety of ways. Related to contract components, some would like to monitor the resource and capability commitments and report violations of those independent of the problem parameter and measurable performance obligations. Others plan to continuously monitor the measurable performance obligations and to selectively monitor the resource, capability, and problem parameter commitments only when the performance obligation is breached.

In addition to the opportunities for experimentation regarding which contract components to monitor, many other tunable monitoring parameters exist, some of which are listed here. The frequency with which contract commitment status is checked is adjustable. Discrete measurements can be considered individually, averaged over a window of time, or averaged over a number of samples. Outlier values can be discarded, averaged in with other measurements, or given no special treatment. Individual commitments can be monitored separately, a central monitor can watch all contract obligations, or a hierarchy of monitors can be set up.

Beyond the mere monitoring of contract commitments lies the questions of when and how to flag contract violations. As was previously mentioned, the variability inherent in the Grid, combined with the inexact nature of most capability and measurable performance commitments, will likely dictate that only those contract violations exceeding some configurable level of severity will be reported. To report every contract violation, using the strict definition of the term, would be overly reactive given natural fluctuations in the system. The structure of the contract monitoring system, be it distributed and independent processes, a single centralized monitor, a hierarchy of processes, or some other architecture, will have a direct impact on how an "overall contract violation" is detected and reported.

One goal of the GrADS project is to use the knowledge that a contract violation has occurred, combined with indications of why the violation arose (resource, capability, problem parameter, or faulty performance model), to reallocate resources and/or improve performance predictions. To achieve this goal, the contract monitoring system must convey to the larger GrADS software infrastructure the contract violation information it has accumulated, allowing the scheduler and the program preparation system to utilize the findings and adjust system resources and performances predic tions when it seems beneficial to do so. In some cases, better resources will not be available and the application will be allowed to continue under the current conditions, even if the contract is continually violated. In other cases, the GrADS system may halt execution if the contract is violated and better resources are not available. And, in other instances, migration to new resources could help improve performance.

These scenarios point to the need for not only the application, but also the contract, to adapt over time. This adaptation could be the result of the application migrating to new resources and causing a rewrite of the contract, or to a phase change in the application that requires new measurable parameter commitments. In some applications, there are distinct phases of computation behavior. The application may move through these phases sequentially, alternate between them in random or regular patterns, follow a route directed by user input or "steering", or take an execution path that is not predictable. While work to date has concentrated on contracts with a single type of behavior for the lifetime of the application, clearly the contract development and monitoring systems must be extensible to accommodate new and/or updated contracts in the course of application execution.

5 Software Infrastructure Supporting Contracts

Previous sections have defined the commitments that make up a GrADS contract and described required characteristics of the contract definition and monitoring process. In this section we will show how the *Autopilot* toolkit supports these efforts in the GrADS execution environment. An overview of the contract monitoring system will be given first, followed by descriptions of the relevant Autopilot components.

5.1 The GrADS Contract Monitoring Architecture

Applications in the GrADS project run on Grid resources and are therefore often placed on widely distributed systems under the control of various organizational units. The contract monitoring system must be able to operate effectively in such an environment.

Figure 1 shows a basic implementation of a contract-monitoring system in the GrADS environment using the Autopilot toolkit. *process 0* and *process 1* represent two processes of a GrADS application. Each process has been instrumented with Autopilot sensors to collect measurable performance readings as the application executes. The application may also have sensors that make available the actual program parameters used in the run, or other values intrinsic to the performance model for the application. When the application is started, the sensor(s) in the instrumented processes register their existence with an Autopilot Manager process running somewhere on the Grid. The Autopilot Manager process can be running at all times, or can be started by the *GrADS Application Launcher*, which starts the actual application.

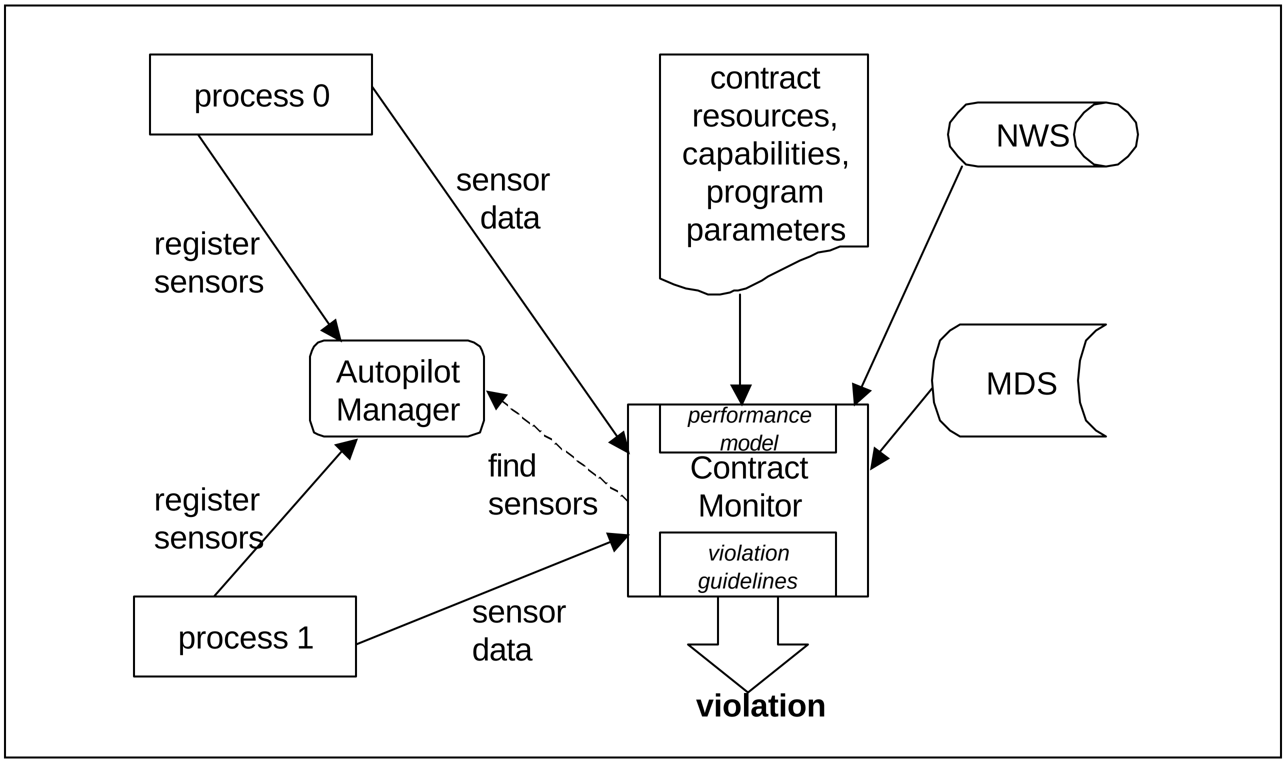

Figure 1: The GrADS Contract Monitoring Architecture

The contract monitor process is also started when the application is launched. It takes as input the contract commitments in the form of resources, capabilities, and program parameters. The performance model for the application that is used to predict the expected measurable performance given the resources, capabilities, and program parameters is included as a callable routine in the contract monitor. Also included in the contract monitor are contract violation guidelines that control the conditions under which a violation will be reported.

When the contract monitor is started, it queries the Autopilot Manager to locate the sensors that will be reporting the measurable performance information for the application. Once the connections between the contract monitor and the sensors are established, the monitor receives sensor data directly from the application processes. This data can be used to verify program parameters and expected execution behaviors used in the model. The sensor data is also used to check application progress as predicted by the performance model for the given resource, capability, and program parameter commitments. In addition, the contract monitor can query MDS and NWS servers to obtain up-to-date resource and capability status. These queries can be initiated by the contract monitor, as shown in Figure 1, or could be implemented via Autopilot sensors associated with the MDS and NWS data. In the latter approach, those sensors would also register with an Autopilot Manager, be "found" by the contract monitor, and periodically send data directly to the contract monitor without ongoing queries on the part of the monitor.

By combining the contract commitments from the resources, capabilities, program parameters, and performance model, and comparing them to the measured values from the application processes and the real-time resource and capability reports, the contract monitor can determine when a contract violation occurs. In the current implementation, the contract monitor for each application is tailored to that application, although the general structure of the contract monitors is very similar. Every contract monitor takes the same types of inputs (commitment specifications and sensor values) and performs the same basic operations (comparison of expected and actual values to detect contract violations). Within these guidelines there is total flexibility as to how often sensor values are read, how they are averaged or ignored, if and when resource and capabilities measurements are checked, and what violation guidelines are implemented.

In the basic architecture shown in Figure 1, a single contract monitor process performs all the monitoring for the GrADS application.

5.2 Multiple Contract Monitoring Processes

In this section we consider a case where multiple contract monitor processes are used to monitor a single GrADS application. Figure 2 shows such an example, with slightly less detail than was given in Figure 1. In Figure 2, a contract monitor process watches application process 0, another watches application process 1, and a third watches the runtime resource and capability measurements. If any of these three monitors detect a contract violation, an overall violation is reported.

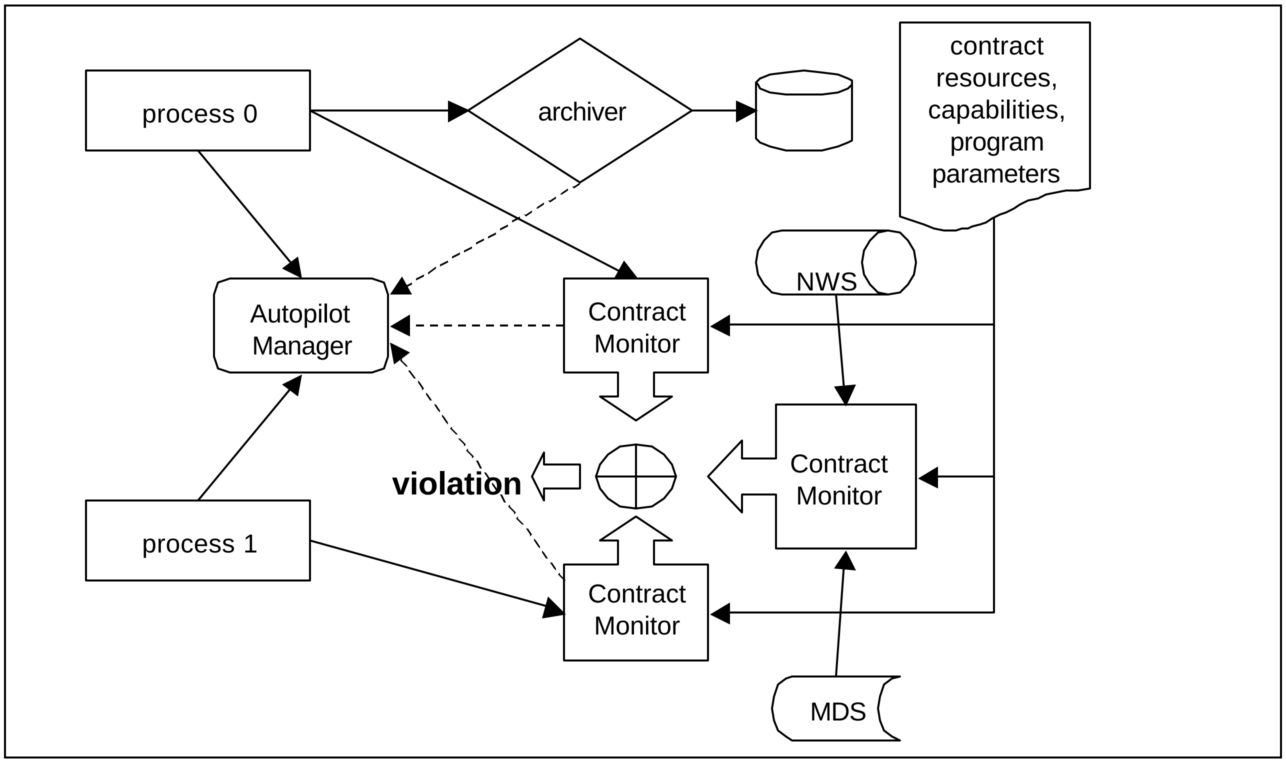

Figure 2: Multiple Contract Monitors

Other possibilities include running multiple contract monitors in parallel, each accepting the same resources, capabilities, program parameters, and sensor values, but implementing different performance models and/or violation guidelines. This can be especially useful in testing various prediction and violation tolerance levels. The Autopilot library supports multiple connections to a single sensor, making this scenario easy to implement.

The support for multiple connections to a single sensor also makes collection and archival of measured performance very straightforward. A separate process, such as the *archiver* process shown in Figure 2, can receive and archive the sensor output for later analysis without interfering with the operation of the main contract monitor(s). In Figure 2, the archiver only saves sensor data from process 0, but it could just as easily also receive data from process 1. The program *CaptureSensorData,* which is part of the Autopilot distribution and available on all GrADS systems, provides a general-purpose capture and archive capability.

5.3 Autopilot Tagged Sensors

Autopilot tagged sensors, which were introduced briefly in the previous sections, provide a mechanism for making data values accessible to processes other than the one where the data values are kept in memory. The processes receiving the data are called sensor clients. In GrADS, contract monitors are sensor clients, as is the archiver in Figure 2. Autopilot tagged sensors can report their data values to connected clients at specific times in the application via a call to the *recordData* method, or at regular intervals via a timer implemented in a separate thread. Both of these methods *push* sensor values to interested clients. Another option is for the client to *pull* the sensor data across as needed.

The Autopilot sensors offer a wide range of possibilities for collecting, buffering, and distributing the application's measurable performance readings. In addition to these distribution features, there is support for filtering and/or smoothing individual data values prior to sending them to the clients. The reader is directed to the *Autopilot User's Manual* for a complete description of the features and use of tagged sensors. In particular, see the chapter on *Tagged Sensors and Tagged Sensor Clients.*

5.4 The Autopilot Manager

The Autopilot Manager provides a directory registration and lookup service whereby sensors can advertise their existence along with lists of attributes, and sensor clients can locate sensors with attributes of interest to them.

For example, in the case of the GrADS contract monitoring system, the application sensors can register with attributes naming the application they are associated with, the measurable performance record they make available, and the MPI rank of the application process they are part of. A contract monitor process that needs performance measurements from the MPI rank 0 process for the application would query the Autopilot Manager to locate the sensor providing that information. This lookup service means the contract monitor processes need not know where in the Grid the application processes are running in order to monitor them.

It is possible, and sometimes desirable, to have multiple Autopilot Managers running at once. Sensors might register with an Autopilot Manager on a 'local' host that is only queried by 'local' sensor clients. This will reduce the volume of sensor traffic across the wide-area and minimize the delay between when a measurement is taken and when the contract monitor has access to the reading.

One scenario involving multiple Autopilot Managers would be for a contract monitor process to be started on a node of every cluster where any of the application processes are running. Each monitor would be responsible for watching the sensors of all the application processes on nodes in its respective cluster. Each monitor would find those processes by querying the local Autopilot Manager for application matches. If a local violation was detected by the contract monitor, the violation could be reported to a 'master' contract monitor process in charge of collecting and correlating information from the many 'local' contract monitors running on the individual clusters taking part in the computation.

One mechanism for passing information between different contract monitors is to create an Autopilot tagged sensor associated with the 'violation' data in each local contract monitor process. These sensors would register with a 'master' Autopilot Manager, and the master contract monitor would connect to the local contract monitor sensors after finding them via the master Autopilot Manager. When any of the local monitors recorded a violation, they would transmit the associated sensor data to the master contract monitor process.

The GrADS Execution Environment is responsible for making sure that all cooperating processes are aware of the location of the Autopilot Managers they should communicate with. The Execution Environment may also start any Autopilot Managers that are not already running, and will start the contract monitor(s) when the application is launched. Each Autopilot Manager process may service requests from multiple GrADS applications and their associated contract monitors.

The Autopilot Manager is part of the Autopilot distribution and is installed on all GrADS systems. Refer to the *Autopilot User's Manual* for further details on this program.

5.5 Autopilot Tagged Sensor Clients

The GrADS contract monitor(s) contain instances of Autopilot tagged sensor clients that are linked to sensors in the application via the Autopilot Manager lookup process. Recorded application data is transmitted from the application process to the contract monitor, where it is compared with the values committed to in the contract.

As discussed in the earlier section on Autopilot tagged sensors, a variety of distribution mechanisms are available including sensor-initiated transmission of measured values (push) and client-initiated requests for readings (pull). Each tagged sensor client instance may connect to one or more tagged sensors, depending on the specificity of the attributes in its query to the Autopilot Manager. If a single sensor client receives multiple types of sensor data, the client code receiving the data must be robust enough to recognize and process multiple types of incoming records.

Whenever sensor data is received, a client handler thread is started to process the data. Since the underlying Globus/Nexus delivery mechanism does not guarantee in-order delivery of messages, and since the handler thread scheduling is not under user control, it is possible that measurement records will not be processed in the order they were produced. Contract monitors need to take this into account, and in some cases a sequence number inserted in the measurement record is helpful in comparing measurements to predictions.

The flexibility of Autopilot tagged sensors and tagged sensor clients allows for many configurable parameters in the implementation of contract specification and monitoring. Frequency of monitoring, smoothing of data, single and multiple monitors are all easily supported. The reader is again directed to the *Autopilot User's Manual* for a full description of the tagged sensor client capabilities.

5.6 Autopilot Decision Procedures

While it is possible to code a contract monitor that explicitly compares commitments with run-time measurements and reports violations when expectations are not met, the uncertainty in commitments, temporal variations in measurements, and the existence of multiple types of commitments which must somehow be combined to reach a final outcome (among other things) can make this approach quite cumbersome. Autopilot provides an alternative through its decision procedure mechanism that is based on fuzzy logic.

The fuzzy logic approach allows one to linguistically state contract violation conditions. The statements "if the ratio of measured flops/second to committed flops/second, referred to as the *flopsRatio*, is BAD, then the *contract* is VIOLATED; if the *flopsRatio* is GOOD, then the *contract* is OK" demonstrate a simple fuzzy logic rule set. Unlike boolean logic values that are either true or false, the fuzzy variables *flopsRatio* and *contract* can assume any value ranging from 0 (false) to 1 (true) with a variety of different transition functions from 0 to 1.

The Autopilot decision procedure input defining these rules and the transition functions for the *flopsRatio* and *contract* fuzzy variables is shown here:

```
rulebase ContractRules;

var flopsRatio( 0, 2 ) {
   set trapez bad ( 0.0, 0.5, 0.0, 0.5 );
   set trapez good( 1.0, 2.0, 0.5, 0.0 );
}

var contract( -0.5, 1.5 ) {
   set triangle violated( 0.0, 0.5, 0.5 );
   set triangle ok      ( 1.0, 0.5, 0.5 );
}

if ( flopsRatio == bad )  { contract = violated; }
if ( flopsRatio == good ) { contract = ok; }
```

Figure 3 shows the transition functions for *flopsRatio*, the input fuzzy variable. The ratio of the measured flops/second, as reported by the application sensor, to the committed flops/second, as committed to in the contract, would map to the x-axis "crisp value". Based on that value, corresponding levels of truth for BAD and GOOD would be determined from the transition functions for each. Say the ratio is .75, then the truth-value for BAD is .5 and the truth-value for GOOD is also .5. Clearly, the transition functions for BAD and GOOD can be adjusted to change the "mapping" for a given flops ratio. These functions are but one control the fuzzy logic decision procedures offer to handle variability in contract commitments and performance measurements.

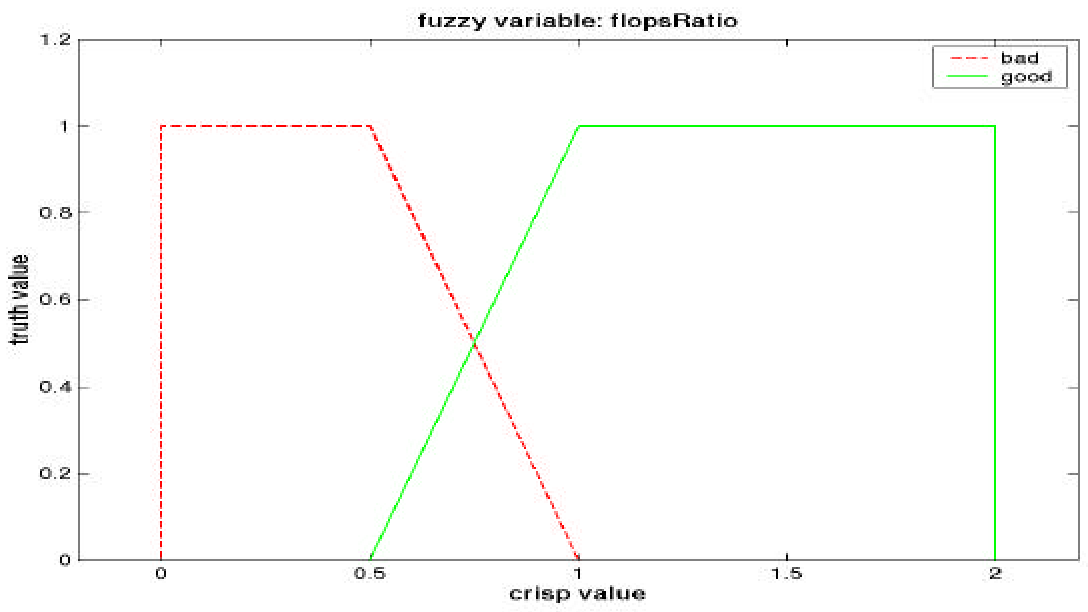

Figure 3: *flopsRatio* Fuzzy Variable Transition Functions

The upper triangles in Figure 4 show the transition functions for *contract*, the output fuzzy variable. Since the antecedents for both rules in the example are partially true, both rules will contribute to the final contract outcome. Both BAD and GOOD have truth-values of 0.5, so the OK and VIOLATED transition function curves shown in figure 4 will be scaled by one-half, yielding a maximum value for each of 0.5 on the truth value y-axis for the crisp input ratio of .75 in this example. These scaled values are shown in the shorter triangles of Figure 4. To compute the final outcome of the contract, the scaled contract truth value curves are combined through a function such as maximum, bounded sum, or center of gravity, and defuzzified into a single crisp contract output ranging from –0.5 to 1.5 in this example.

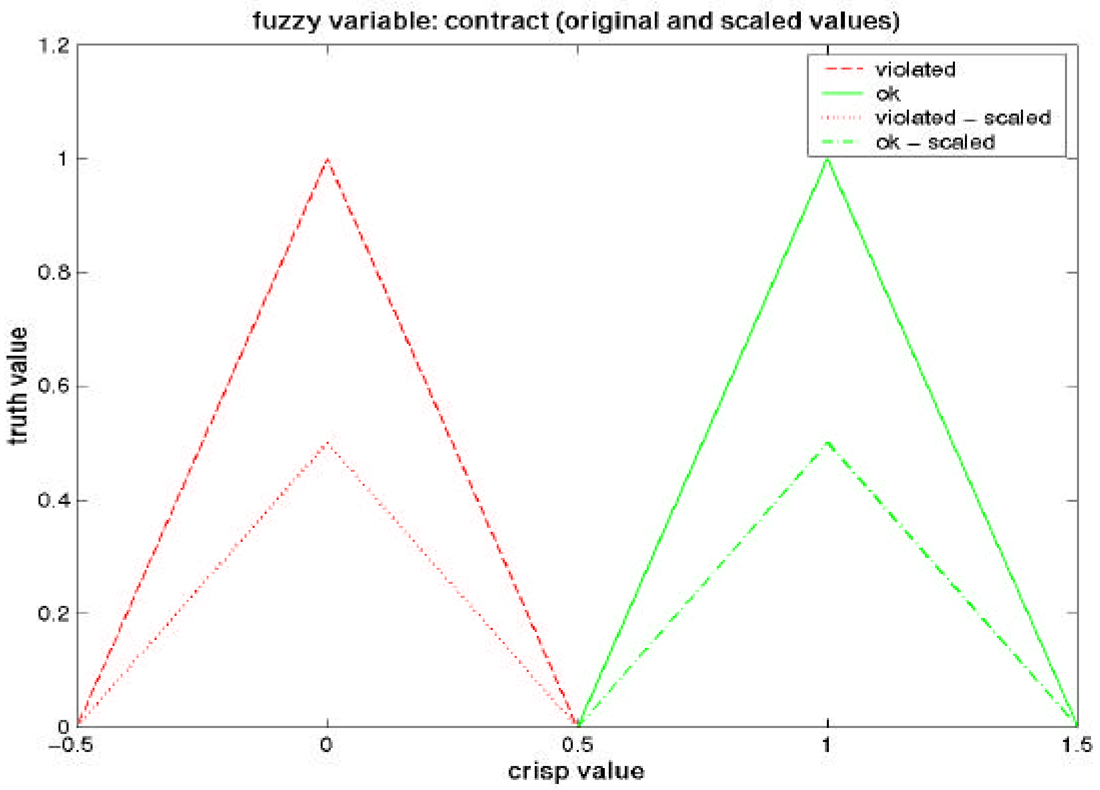

Figure 4: *contract* Fuzzy Variable Transition Functions

Figure 5 shows the results of this fuzzy rulebase for a variety of committed and measured floating-point operations per second. Focus, for example, on the case where 30 FLOPS are committed by the contract. When the measured FLOPS are less than 15, the contract output is 0. As the measured FLOPS increase to 30, the contract output smoothly increases to slightly over 1. Intuitively, the contract becomes truer as the measured FLOPS get closer to the committed FLOPS.

By adjusting the rules and fuzzy variable transition functions, the user has much control over the contract output. The contract output value can be "processed" even further, perhaps to only report "severe" violations (for example, contract output less than .3) when the scheduler indicates the Grid resources are heavily loaded, and less "severe" cases (contract output less than .6 perhaps) when there are many idle resources and successful rescheduling would be more likely.

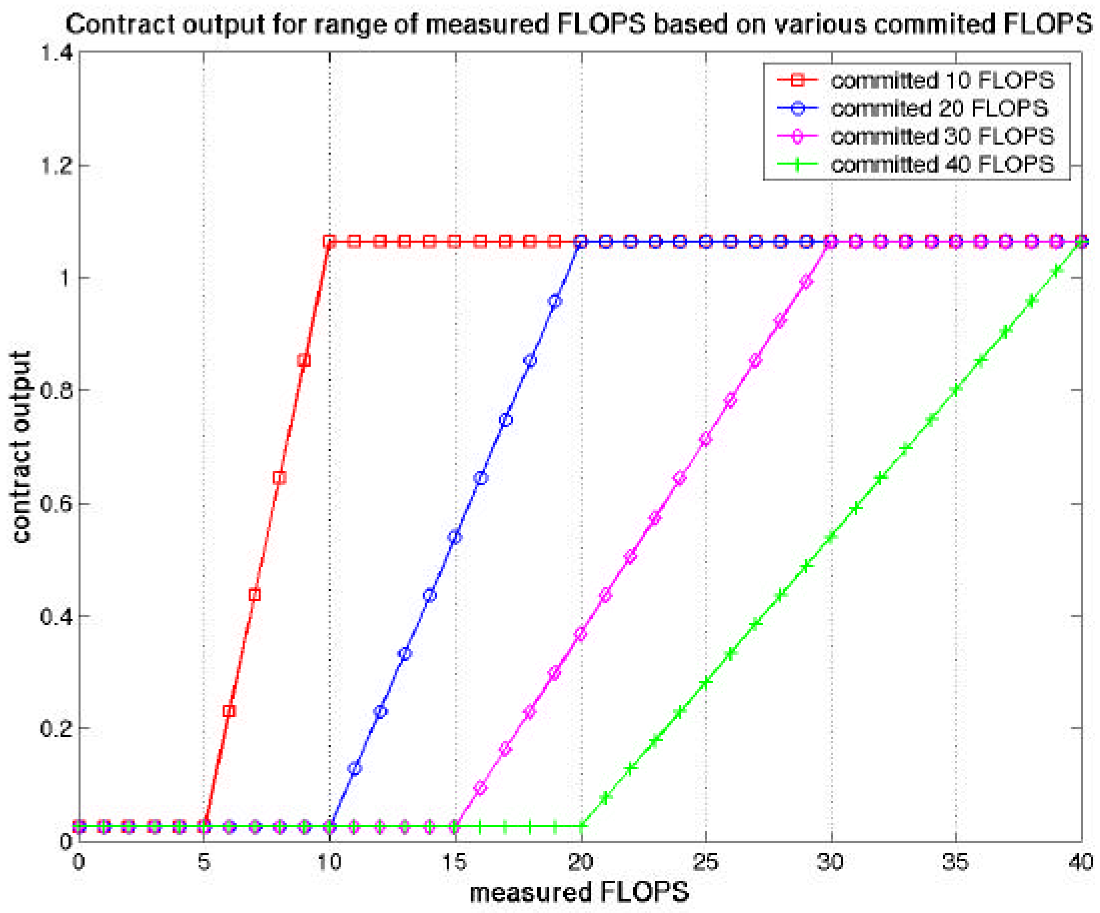

Figure 5: Range of Contract Outputs for Various Measured to Committed FLOPS Ratios

The Autopilot decision procedures with their support of fuzzy logic rules offer many features that are attractive in the GrADS environment. For a more complete discussion of fuzzy rule bases and the Autopilot classes that support these decision procedures, please refer to the *Autopilot User's Manual.*

6 GrADS Contracts for a ScaLAPACK Program

In this section we discuss two GrADS contracts for *PDSCAEX*, a parallel program that uses ScaLAPACK library routines to perform LU factorization of a distributed matrix. The first contract uses a performance model based on developer knowledge of application behavior, while the second contract uses historical data to formulate the contract. The contract specification and monitoring for these two contracts are presented to illustrate actual implementations that make use of some infrastructure features described earlier.

6.1 A Contract Based on Developer Knowledge of Application Behavior

The first PDSCAEX contract takes advantage of developer knowledge of application behavior to model expected measurable performance based on a given set of resources, capabilities, and problem parameters.

6.1.1 Resources

The *PDSCAEX* program executes on some number of processors located on the Grid. These processors constitute the resources that are committed to in the contract. Implicitly, the networks connecting the processors are also resources that are committed to. The resources are specified as a vector of hostnames, with the vector indicating the assignment of MPI processes to hosts. The MPI processes implement the parallel algorithm for the application, and it is possible that more than one MPI process will be assigned to a single host processor.

Resources = < host1, host2, … hostP >

6.1.2 Capabilities

For the contract based on developer knowledge of application behavior, the capabilities include a vector indicating expected flops/cycle for each of the processors in the resource list. These expected flops/cycle values were derived from a combination of the processor MHz rates (Pentium processors) and the load on the individual processors at the time the job was submitted. The load is provided by NWS and gives the fraction of time-shared CPU cycles that are available for approximately the last 10 seconds prior to job submission.

In addition to the flops/cycle capability for the processors, latency and bandwidth expectations between pairs of processors are also specified. These network capabilities are given as two vectors, each with P*P elements, representing the network connections between pairs of processors. These vectors include entries for the "connections" of each processor to itself, and those entries are set to –1. The latency measurements (lat) are expressed in milliseconds, and the bandwidth (bw) in megabits/second. The network capabilities are provided by NWS.

Capabilities = < flopsPerCycle1, flopsPerCycle2, … flopsPerCycleP >;
< lat1-1, lat1-2, … lat1-P, lat2-1, lat2-2, … lat2-P, …, latP-1, latP-2, … latP-P >;
< bw1-1, bw1-2, … bw1-P, bw2-1, bw2-2, … bw2-P, …, bwP-1, bwP-2, … bwP-P >

6.1.3 Problem Parameters

Three problem parameters are specified in the contract. These are the matrix size (N), the panel size (NB) that specifies the number of matrix columns allocated to each process in the algorithm, and the number of processes (P). Note that P, the number of processes, is the same P as in the resources and capabilities commitments.

Problem Parameters = < N, NB, P >

6.1.4 Measurable Performance

The measurable performance commitment for the contract based on developer knowledge of application behavior states that iteration I of the main application loop will be completed in T seconds. A developer-provided performance model, which uses the resources, capabilities, and problem parameters as inputs, predicts T for each iteration I.

Measurable Performance T(I) = performance_model(Resources, Capabilities, Problem Parameters, I)

6.1.5 Contract Monitoring and the Execution Environment

The GrADS Program Preparation and Program Execution Systems are not yet fully functional. For that reason, some things were done manually for the *PDSCAEX* application that we expect to be automated in the future. The basic steps required to instrument, launch, and monitor the application should be very similar. Those steps are outlined in this section.

- Prepare the application to report runtime performance measurements that can be compared to the measurable performance commitments:

  - Routines were written in C++ to perform Autopilot startup tasks, create and register a tagged sensor, record sensor values, and perform Autopilot shutdown tasks. These routines, which are callable from the Fortran application, constitute about 150 lines of code. The startup and shutdown routines will be almost identical for any application. Routines to create sensors and record sensor values depend on the specific information that is being collected, but the basic code structure can be reused.

    For the *PDSCAEX* application, the sensor records the MPI rank of the process, the iteration number, the timestamp, the elapsed seconds since the previous measurement, and the hostname.

  - The application source code was hand-instrumented to call the Autopilot startup and shutdown routines. A call to the routine that records the sensor data once per iteration was inserted in the main loop.

- Create a contract monitor to receive runtime application performance metrics and check for contract violations:

  - A program was written in C++ to perform contract-monitoring duties. This ContractMonitor program reads an input file that contains the hostname of the Autopilot Manager, as well as the resources, capabilities, and program parameter commitments for this run of the application. The entire program contains approximately 300 lines of code.

  - The ContractMonitor includes the developer-provided performance model that gives the expected duration T for a given loop iteration I. This model is encapsulated in a single routine that is about 80 lines of code. The model takes into account the number of floating point operations, messages sent, and bytes transferred for a given iteration, using the capabilities commitments for the processor and networks to calculate the expected duration.

  - A fuzzy logic rule base gives the conditions under which the contract will be deemed violated. For this application, the violation is based on the ratio of the measured iteration time to the expected iteration time. When the ratio is low, the contract is okay. The rule base is given here:

```
rulebase Grads_Contract;

var timeRatio( 0, 10 ) {
   set trapez LOW  ( 0,  1, 0, 1 );
   set trapez HIGH ( 2, 10, 1, 0 );
}

var contract( -1, 2 ) {
   set triangle OK       ( 1, 1, 1 );
   set triangle VIOLATED ( 0, 1, 1 );
}

if ( timeRatio == LOW  ) { contract = OK; }
if ( timeRatio == HIGH ) { contract = VIOLATED; }
```

    This rule base is compiled into an object file that is linked in with the ContractMonitor program and accessed via the rule base name (Grads_Contract).

  - At runtime, the ContractMonitor code instantiates an Autopilot tagged sensor client that locates the application sensor via the Autopilot Manager process and connects to the located sensor. When the application records sensor data for a loop iteration, that data is received by a handler function in the ContractMontior process. The handler reads the sensor data values, prints them to standard error, and calls a "makeDecision" routine. The makeDecision routine computes the ratio of measured to expected iteration duration and invokes the Autopilot fuzzy logic decision procedure that sets the contract value. Based on the allowable threshold for that value, the contract monitor may report that the contract has been violated.

    The code to instantiate the sensor client and handle the incoming data is approximately 130 lines long. The makeDecision routine is 20 lines of code, including debugging output.

  - Although many details of the Contract Monitor code are specific to this application, the general framework of the program and routine structure should be reusable with other applications.

- Launching the Autopilot Manager, *PDSCAEX* application, and contract monitor:

  - An Autopilot Manager process is started manually on a given host prior to application launch.

  - The application is launched with an argument specifying the host where the Autopilot Manager is running.

  - The contract monitoring process is launched at the same time as the application, with an argument specifying the input file containing the runtime configuration information.

- Contract monitoring:

  - In the current implementation, the ContractMonitor process only connects to the sensor for the application process with MPI rank 0. This could be extended by having this single monitor attach to other application processes, or by having additional monitor processes launched.

  - The ContractMonitor process bases its contract violation decision on the comparison of the measurable performance commitments with the actual measured iteration durations. At this time it does not try to verify that the resource, capability, and program parameter commitments are being met, or that the expected execution behavior underlying the performance commitment is accurate.

  - The ContractMonitor process reports the contract violation by writing to standard error. In the future, we expect this information to be sent to a rescheduling process.

  - The current implementation does not attempt to identify the cause of a contract violation. Preliminary results indicate that the developer-provided performance model needs additional tuning for use in the Grid environment with its high-latency/low-bandwidth network links. This work is currently underway.

This example demonstrates the mechanisms for specifying and monitoring a GrADS contract for the *PDSCAEX* application using the Autopilot toolkit components integrated into the application, the contract monitoring process, and the runtime environment.

6.2 The History-Based Performance Model

The second PDSCAEX contract uses *application intrinsic* knowledge gathered from a previous execution combined with resource capabilities to model expected measurable performance. Because the contract commitments are based on information from a previous execution of the application, we refer to this contract as using a history-based performance model.

6.2.1 Resources

Resources are the same as for the first PDSCAEX contract.

Resources = < host1, host2, … hostP >

6.2.2 Capabilities

For the history-based contract, the capabilities include a vector indicating expected flops/second for each of the processors in the resource list. These expected flops/second values were derived from various sources for different runs of the application. In one case, the flops/second capability for a given processor was the average flops/second the application achieved on that processor in a previous run. In a second case, the value used for a processor was the maximum flops/second achieved by a benchmark program running on the processor, scaled by the NWS load at the time the job was submitted. In a third case, the capability used was half of the flops/second value used in the second case.

In addition to the flops/second capacity for the processors, bandwidth expectations in bytes/second for each processor are also specified. Note that in contrast to the first PDSCAEX contract where bandwidth capacities between pairs of processors are given, this contract uses a bandwidth capability specification for each processor. Latency capabilities are not specified in the second contract. The bandwidth network capabilities are given as a vector with P elements.

As with the processor capabilities, the network bandwidth expectations were derived from various sources for different runs of the application. In the first case, the bytes/second capability for a given processor was the average bytes/second the application communicated from/to that processor in previous run. In a second case, the processor bandwidth capability was set to the minimum bandwidth of any link connected to the processor, as provided by NWS. In a third case, half of the minimum NWS-reported bandwidth for links to the processor was used.

Capabilities = < flopsPerSec1, flopsPerSec2, … flopsPerSecP >;
<bytesPerSec1, bytesPerSec2, … bytesPerSecP >

For the examples shown in this document, the expected flops/second and bytes/second were based on averages achieved by the application in a previous run on the same resource set. This corresponds to the first of the three cases outlined in the preceding paragraphs on flops/second and bytes/second capabilities.

6.2.3 Problem Parameters

Problem Parameters are the same as for the first PDSCAEX contract.

Problem Parameters = < N, NB, P >

6.2.4 Measurable Performance

The measurable performance commitment for the history-based contract states that for any iteration of the main application, the measured messages/second and instructions/second will fall within a certain range of a predicted messages/second and instructions/second, where the prediction is based on the historic application intrinsic knowledge combined with the resources, capabilities, and problem parameters.

Measurable Performance (Messages/Second,Instructions/Second) =
performance_model(Resources, Capabilities, Problem Parameters, Historical_Data)

To obtain the application intrinsic parameters used in the history-based performance contract, PDSCAEX was run with the same Problem Parameters on processors of the same architectural family. The program was instrumented with PAPI to collect instruction count and floating point operation count, and with a Pablo-developed MPI profiling library to collect message count and bytes transferred. These counts, which are constant for all executions of PDSCAEX for the given Problem Parameters on processors of this architecture, were reported for each iteration of the main loop via Autopilot tagged sensors.

The general-purpose Autopilot program *CaptureSensorData* was used to collect the sensor output and save it to a file. From this collected data, the metrics "instructions/flop" and "messages/byte" were computed for all loop iterations. The resulting points were plotted to yield an *application signature* showing the relative mix of computation and communication for the PDSCAEX application with the given Problem Parameters and architecture. The application signature for MPI node 0 of PDSCAEX on the Intel x86 architecture with Problem Parameters < N = 2000, NB = 64, P = 4 > is shown in Figure 6. Metrics for the last four loop iterations, numbers 29-32, are not shown in the plot as those points were clearly outliners and skewed the axes scaling factors.

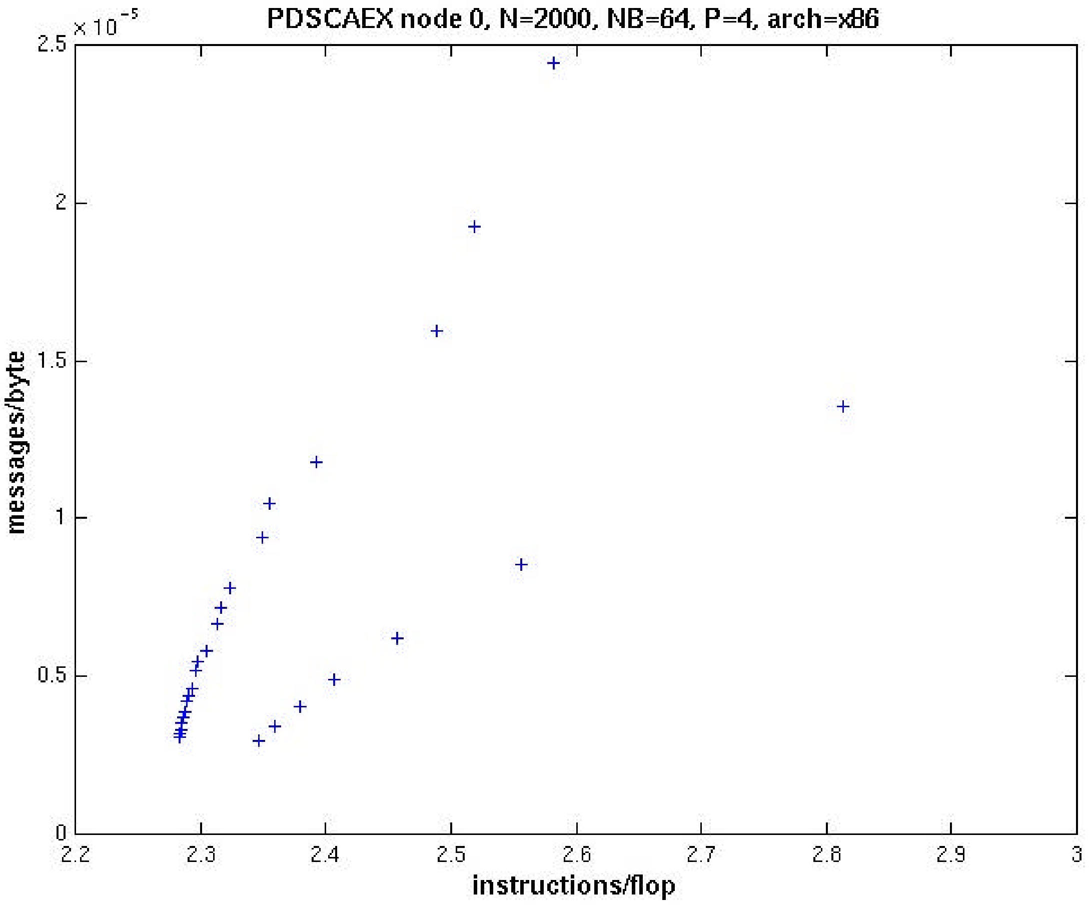

Figure 6: Application Signature for PDSCAEX

In the future, we anticipate building a repository of application signatures for different problem parameter and processor architecture combinations. We believe it may also be possible to generate such a signature in real-time for long-running applications, where the signature will evolve as the application runs. We also hope the compiler in the GrADS PPS may be able to provide application-intrinsic metrics, such as statement count, which are not processor architecture-dependent like the current instruction count metric.

Once the application signature is available, it is projected onto a measurable performance space using the contract flops/second and bytes/second capabilities. This projection yields a new measurable performance space with axes of instructions per second and messages per second:

Application Intrinsic		**Contract Capabilities**		**Measurable Performance**
$\frac{\text{instructions}}{\text{flop}}$	x	$\frac{\text{flops}}{\text{second}}$	=	$\frac{\text{instructions}}{\text{second}}$
$\frac{\text{messages}}{\text{byte}}$	x	$\frac{\text{bytes}}{\text{second}}$	=	$\frac{\text{messages}}{\text{second}}$

At this stage, the projected measurable performance signature points were clustered using a standard clustering algorithm, with outliers discarded. For Contract Capabilities of 111.3 Mflops/second and 2.91 Mbytes/second the process yielded three clusters as shown in Figure 7.

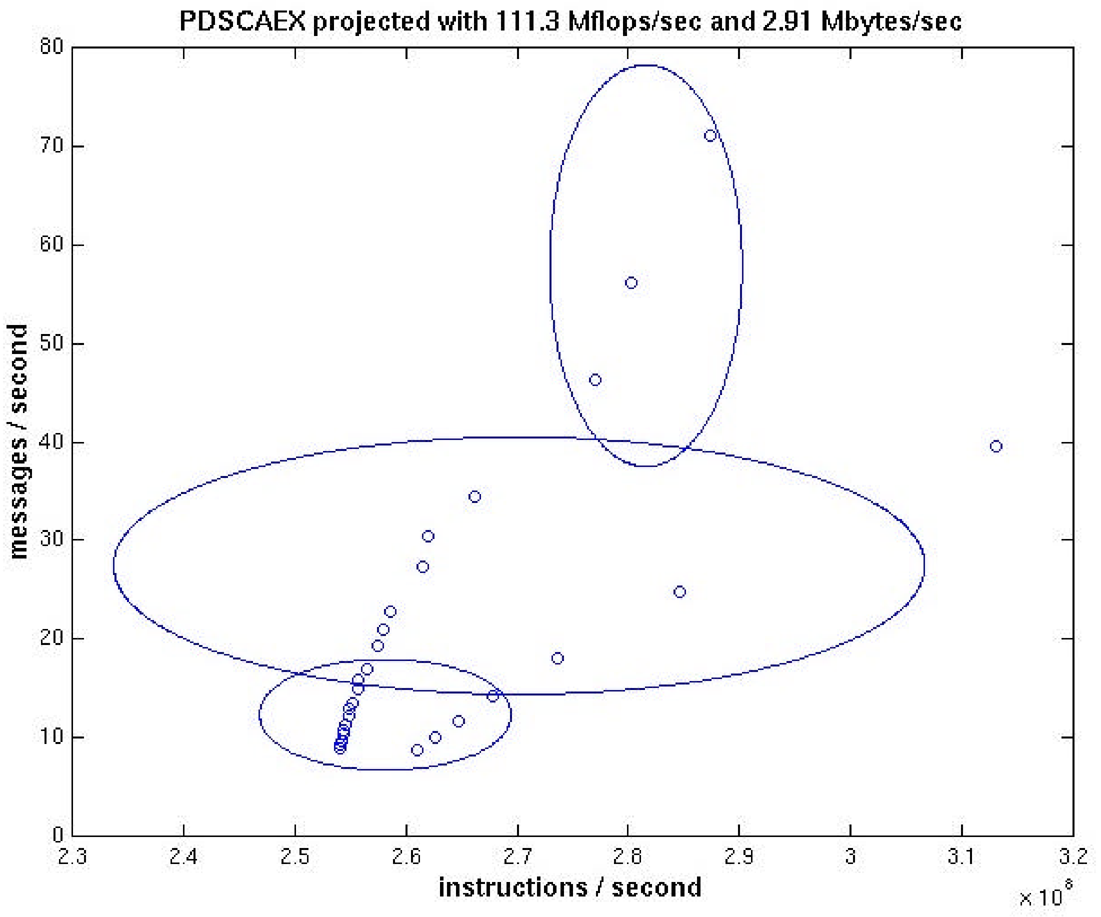

Figure 7: Projected Measurable Performance Signature for UIUC opus machines (4)

Capabilities: 111.3 Mflops/sec; 2.91 Mbytes/sec

Figure 8 shows the projected measurable performance signatures for three different sets of contract capabilities based on historical data for three different virtual machines. The projected measurable performance signature in Figure 7 is also shown in Figure 8 -- it is the signature falling farthest to the right on the x-axis.* The signatures for the other two virtual machines, each with slower processor and network capabilities, appear to the left of this signature. You see clearly that the expected measurable performance varies depending on the capabilities of the resource set (virtual machine) used. The range of expected messages/second and instructions/second is widest for the virtual machine with the fastest processor and network capabilities.

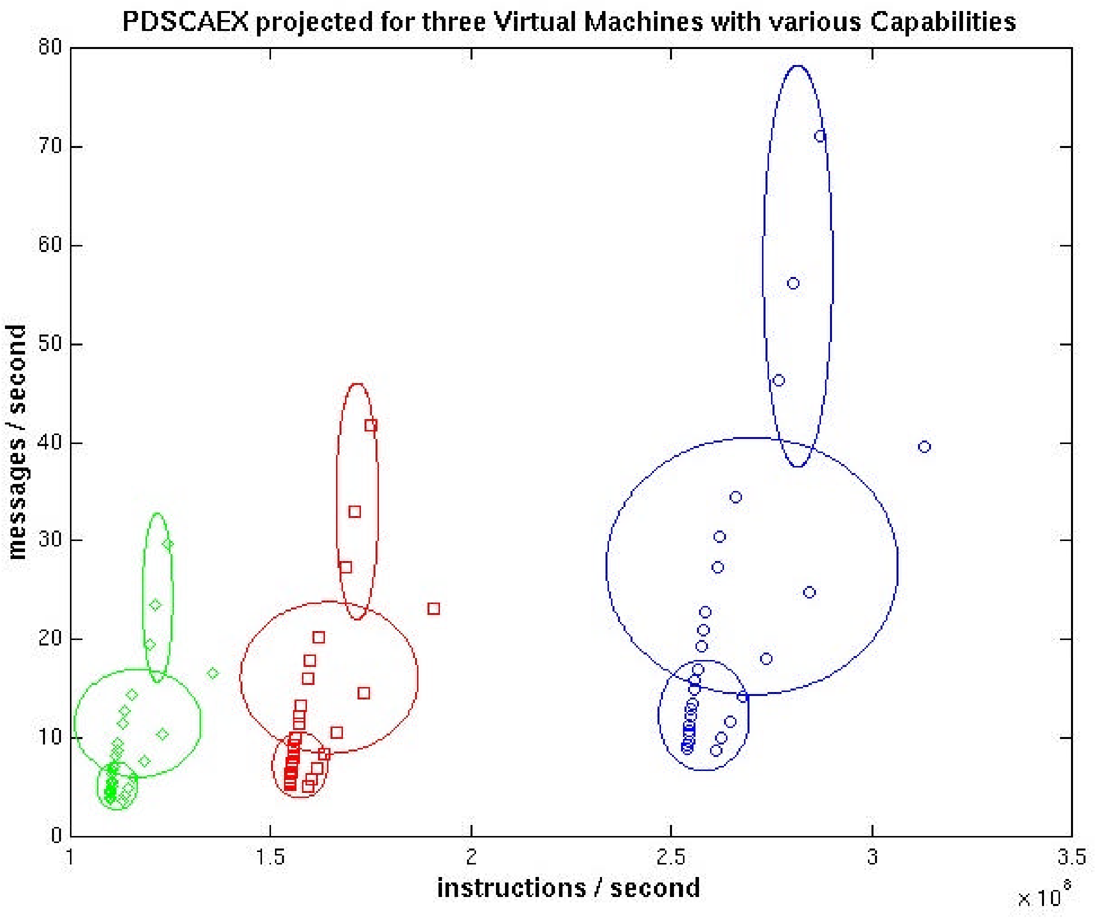

Figure 8: Projected Measurable Performance Signatures for three Virtual Machines
1) **[green diamonds] UIUC major machines(2) UTK torc machines(2)**
Capabilities: 48.1 Mflops/sec, 1.22 Mbytes/sec
2) **[red squares] UIUC major machines (4)**
Capabilities: 67.8 Mflops/sec, 1.71 Mbytes/sec
3) **[blue circles] UIUC opus machines (4) with Myrinet**
Capabillities: 111.3 Mflops/sec, 2.91 Mbytes/sec

* Note that Figures 7 and 8 have different axes limits.

As noted earlier in the discussion of capabilities found in section 6.2.2, it is possible to choose from a variety of capability projections, even for the same resource set. More experiments are needed to determine the most appropriate capabilities specifications in the Grid environment.

6.2.5 Contract Monitoring and the Execution Environment

The mechanisms for preparing the application to report runtime performance measurements and creating the contract monitor to receive runtime application performance metrics and check for contract violations are identical to those outlined in Section 6.1.5 for the first PDSCAEX contract.

For the history-based performance model, the contract monitor computes the normalized distance of the reported runtime metrics (instructions/second, messages/second) from the closest cluster centroid in the measurable performance signature for the virtual machine where the application is executing. The fuzzy logic rule base encapsulates the notion that the farther the measured point is from one of the expected clusters, the more the contract is violated:

```
rulebase Grads_Contract2;

var distanceFromCentroid(0, 100) {
   set trapez SHORT ( 0,   4, 0, 4 );
   set trapez LONG  ( 8, 100, 4, 0 );
}

var contract( -1, 2 ) {
   set triangle OK       ( 1, 1, 1 );
   set triangle VIOLATED ( 0, 1, 1 );
}

if ( distanceFromCentroid == SHORT  ) { contract = OK; }
if ( distanceFromCentroid == LONG ) { contract = VIOLATED; }
```

As with the previous rulebase, adjusting the definitions of the fuzzy variables allows different margins of tolerance in the evaluation of acceptable behavior.

To date, the majority of the effort has gone into identifying appropriate application intrinsic metrics and capabilities specifications, and the clustering and contract monitoring is currently being done off-line. However, with techniques already in use in other projects, we anticipate being able to complete the entire process in real-time and to not only identify contract violations, but also to zero in on the cause of the violations.

7 Interfaces to the Contract Monitor

Efforts to date have focused on identifying and implementing the components of the contract monitoring system "by hand". As we gain more experience, and as the GrADSoft architecture matures, the various interfaces and parameters to the contract monitoring system that are currently defined on a per-application and per-contract basis will become more general and well-defined. In this section we identify some of these parameters and interfaces, and offer preliminary thoughts on how they might be standardized in the GrADSoft architecture.

Note. In the next sections I use Italics to highlight areas where I am not confident of the mapping I speculate about to the GrADSoft architecture. Stay tuned as further discussions with GrADSoft folks help this part evolve -- Ruth

7.1.1 Specification of Contract Commitments

The resource, capability, problem parameter, and measurable performance commitments of the contract must be conveyed to the contract monitor.

Currently there is an implicit agreement between the contract writer and the contract monitor as to how these commitments will be passed. The measurable performance commitments are "compiled in" via the performance model used to predict the measured values, and the sensors to capture those values are inserted by hand in the application. The contract monitor code explicitly locates relevant sensors and passes data of interest to the fuzzy logic rulebase (when one is used).

We believe that there will be some variation from application to application in types of resources and capabilities listed, but that the union of all types will be fairly small. The problem parameters are necessarily application specific. There should be a standard mechanism for specifying the lists of resources, capabilities, and problem parameters – even if the type of information in those lists varies from application to application.

It seems the resources are encapsulated in the virtual machine and in the output of the Mapper. Capabilities available via Grid Information Repository – are they also available in the Application Manager or some other place for the particular virtual machine where the problem will execute? Problem parameters seem to be available in the GrADSoft problem (versus the application). In the case where historical data is used, it should be available via the Grid Information Repository.

Having a "plug in" performance model that predicts run-time monitoring data exported by sensors embedded in the application code allows for a great deal of flexibility as various models are experimented with. We believe it may be possible to develop a configuration or template file that will associate particular sensor values with particular fuzzy logic variables, reducing the coding effort required in the contract monitor. This reduction will not advance the understanding of contract feasibility, but will make it easier to try new things.

This may be the GrADSoft Performance Model. However, we are not concerned with the use of the model prior to the actual program execution… only in the context of the contract monitoring. We hope the PPS will insert relevant sensors in the code to match up with what the performance model requires as input. In the GrADSoft writeup it looks like the VM is encapsulated in the Performance Model so the resources, capabilities, problem parameters may already be in there. Note that we distinguish between the performance model and other parts of the contract monitor (like the fuzzy rulebase, violation thresholds, etc). The model just gives us the measurable performance prediction – what we do with it is another issue.

7.1.2 Contract Monitoring Architecture

As noted earlier, it is possible that there will be one or many contract monitors for a given application. The GrADSoft architecture needs to support

- Specification of one or more Autopilot Managers
- Coordination between Application (and possibly Capability) sensors and the Autopilot Managers they register with, and the lookup of those sensors by contract monitor process(es)
- Launching and coordination of possibly multiple contract monitors and the commitments they are responsible for monitoring.

I believe the Application Manager and Executor will coordinate in much of this effort.

7.1.3 Contract Monitor Specifics

In addition to the contract commitments and performance model discussed above, the contract monitors may have fuzzy logic rule bases, violation thresholds, data smoothing and outlier handing guidelines, and other data handling and decision making parameters associated with them. At this point it is not clear if a general representation for these parameters can be defined, as our set of applications and monitors is limited. We recommend reusing the general contract monitor code framework but hand-coding these parameters at this time. As we gain experience, opportunities to define general interfaces for some or all of these may become apparent.

7.1.4 Reporting Contract Violations

Currently the output of the contract monitor is simply reported via messages standard error. We anticipate making this available information, along with more extensive data about the possible cause of a violation (which commitments were not fulfilled) via Autopilot sensors for use by the resource scheduler system. We believe it may be possible to develop a standard mechanism for relating commitment violatio ns to the original commitment lists, but have not explored this further at this time.

This material is based upon work supported by the National Science Foundation under Grant No. 9975020. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.