CORE
Metadata, citation and similar papers at core.ac.uk
Provided by NASA Technical Reports Server

1 - 51

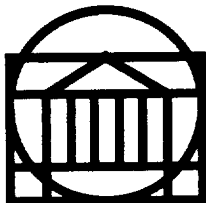
Mentat/A: Medium Grain Parallel Processing

Final Report

(NASA-CR-190891) MENTAT/A: MEDIUM GRAIN PARALLEL PROCESSING Final Report, 1 Sep. 1990 - 31 Aug. 1992 (Virginia Univ.) 51 p
N92-34200
Unclas
G3/62 0121340

Submitted by:
Andrew S. Grimshaw
Principal Investigator

For period: 9/1/90 - 8/31/92
Grant No. NAG-1-1181



DEPARTMENT OF COMPUTER SCIENCE

UNIVERSITY OF VIRGINIA
THORNTON HALL
CHARLOTTESVILLE, VIRGINIA 22903-2442
(804) 982-2200 FAX: (804) 982-2214

Mentat/A: Medium Grain Parallel Processing

Final Report - NAG-1-1181

October 5, 1992

1.0 Overview

The objective of this project is to test the ATAMM firing rules using the Mentat run-time system and the Mentat Programming Language (MPL). A special version of Mentat, Mentat/A (Mentat/ATAMM) was constructed. This required changes to: 1) modify the run-time system to control queue length and inhibit actor firing until required data tokens are available *and* space is available in the input queues of all of the direct descendent actors, 2) disallow the specification of *persistent* object classes in the MPL, and 3) permit only decision free graphs in the MPL. We have been successful in implementing the spirit of the plan, although some goals changed as we came to better understand the problem. In this paper we report on what we accomplished and the lessons we learned.

The remainder of this report is in four sections. Section 2 will discuss the Mentat/A run-time system. Section 3 will briefly present the compiler. In Section 4 we present results for three applications. we conclude with a summary and some observations. Appendix A contains a list of technical reports and published papers partially supported by the grant. Appendix B contains listings for the three applications.

The use of brand names is for completeness and does not imply a NASA endorsement.

2.0 Mentat/A Run-Time System Status

Run-time system support for Mentat/A consists primarily of restrictions that support the ATAMM firing rules, in particular the input queue length restrictions. Below we discuss protocol and implementation changes required for ATAMM firing rule implementation. We then discuss the tracing capability.

2.1 ATAMM Protocol Support

We have made a few minor changes in our implementation plan as described in the proposal. We did this because on closer inspection better techniques were available.

We have created a hybrid type, a *bound regular object*. A bound regular object has a *name* and may have tokens sent specifically to it. This was necessary to support the ATAMM firing rules. Mentat regular objects are created on-the-fly whenever matching tokens are available. Using on-the-fly creation would trivially satisfy the ATAMM rules but would violate the *spirit*. Bound regular objects behave just as ATAMM nodes. Hence all objects in Mentat/A are bound regular objects, and there exists an instance of a bound regular object for each ATAMM node.

Each instance of a bound regular object corresponds to a single ATAMM node in the program graph. Each bound regular object is a master that invokes slave regular objects to perform the actual work. Multiple instances of the corresponding regular objects may be executing at any given time, this corresponds to the desired semantics for later versions of ATAMM. This is illustrated in Figure1 below.

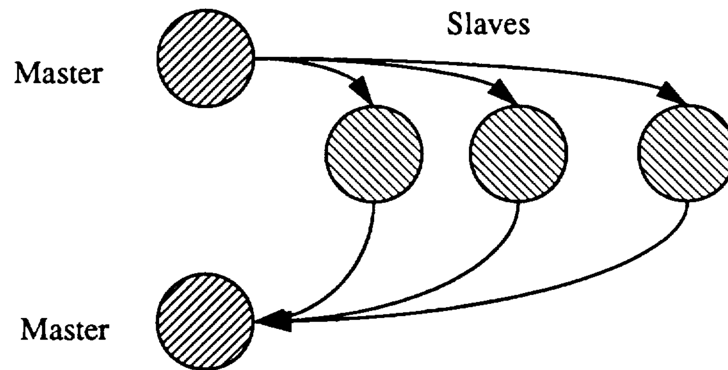


Figure 1. Bound regular masters and regular slaves.

Given bound regular objects, our algorithm for enforcing the ATAMM firing rules consists of two halves, a "node" part, and a "descendent" part. (The firing rules are enforced for bound regular objects only, not for the regular object slaves.) The node part is executed on behalf of a node before it may fire. The descendant part is executed on behalf of each node that is a descen-

dant of the node in question. This is illustrated in Figure2 below.

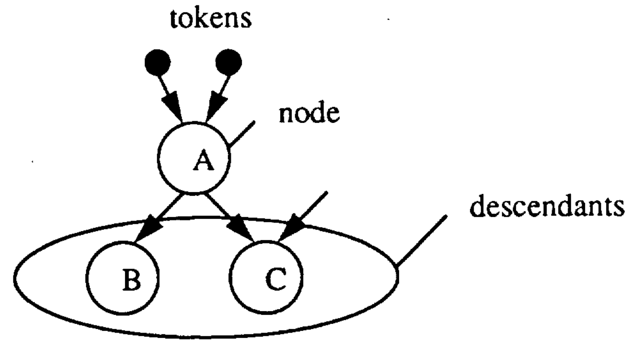


Figure 2. Nodes and Descendants.

The node algorithm for enforcing the ATAMM firing rules is:

1) Match tokens
2) Query all descendants (ATAMM_QUERY), B&C in Figure 1.

    An ATAMM_QUERY consists of the node name and a description of the token that is to be sent.
3) Set *pending_destinations* to number of descendants (2)
4) On receipt of a matching ATAMM_REPLY decrement *pending_destinations*
5) When *pending_destinations* ==0, ready to fire.

Before we can describe the descendant algorithm we must define *work_unit* and *wake_up* lists. A work_unit consists of:

1) a computation_tag - used for token matching,
2) the operation to be performed,
3) the tokens that have arrived,
4) a count of pending tokens,
6) a pending_destination count.

Work units are kept in lists, one list for each operation that can be performed on the object. The *wake_up* list is a list of ATAMM nodes that are waiting for a queue slot, i.e., they are blocked pending a queue slot. Each entry contains the name of the waiting object, and the computation tag, arc number, and operation number of the desired computation. The descendant algorithm when an ATAMM_QUERY arrives is:

1) Find the appropriate work_unit list.
2) See if the potential token "will fit".

3) If so, send an ATAMM_REPLY back to the requestor.

4) If not, enqueue the query in the *wake_up_list*.

5) Check the *wake_up_list* every time the node fires and consumes tokens. If a suspended query "will fit", send the reply.

Thus, the "will fit" decision is the key. It is where we control the queue depth and any other criteria that we may choose to impose. "Will fit" checks the queues to see how many outstanding tokens from the sender are enqueued in the queue for the destination arc-operation. Queue limits may be set on an object-by-object basis using the "set_atamm_queue(queue_depth)" call. Figure3 below illustrates the "will fit" function. If the queue depth has been set to four or more then there

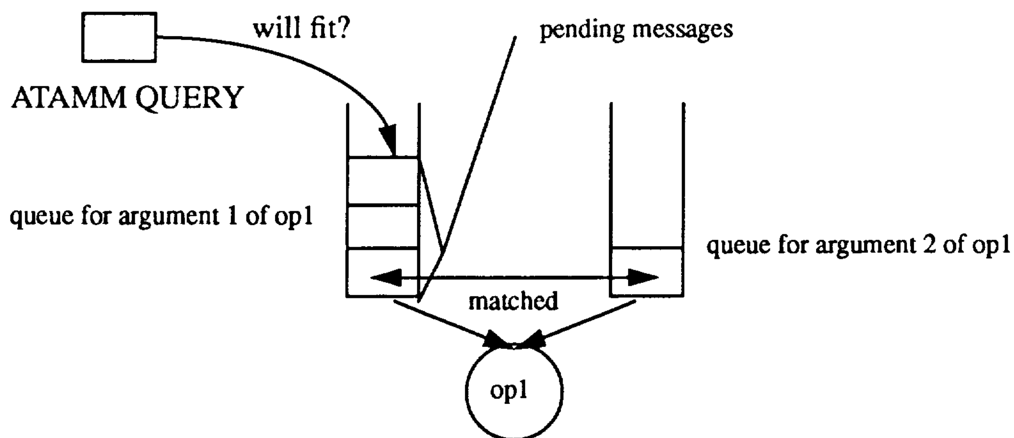


Figure 3. Checking "will fit" for an operation.

is room for the token, an ATAMM_REPLY will be sent, and a placeholder for the token will be created in the queue. If there is no room (maximum queue depth is three), then an entry will be made in the wake up list that contains the name of the node that sent the query.

In addition to the above we must also check the ATAMM firing rules whenever a remote member function is invoked. This is accomplished by sending an ATAMM query for each argument of the member function. The query is handled at the destination as are all other ATAMM queries.

All of the above extensions have been made to the Mentat run-time system. They are enabled by setting the -DATAMM flag when the run-time system is compiled. By using an "ifdef" compiler switch we ensure that Mentat/A benefits from all of the enhancements made to the stan-

dard Mentat and that Mentat/A will execute on all of the same platforms.

2.2 Run-Time System Tracing

We have implemented a tracing capability. When enabled, objects (nodes) log certain classes of events and their corresponding data to a *log_object*. The *log_object* then copies the events to a log file for later interpretation. Each log entry contains the length of the entry, an event tag, a timestamp, the name of the object (node) where the event occurred, the computation tag of the event, and some event specific data. The types of events are:

1) Accept invocation (node fires)
2) Work_unit match (may fire)
3) Token arrival
4) Subgraph elaboration (a copy of the subgraph is included)
5) Operation complete (node complete)
6) ATAMM_QUERY arrival
7) ATAMM_REPLY arrival

Events are stored in the log file. Because of message delays, the order in which they are stored may not be the order in which the events actually occurred. (See Leslie Lamport's classic paper "Time, Clocks, and the Ordering of Events in a Distributed System", CACM July 1978, for a very good description of this problem.) Further, the clock skew on the individual processors may distort the time sequence as observed in the timestamps.

3.0 The MPL/A Compiler

Previous versions of the MPL compiler have been based upon the AT&T 1.2 C++ compiler. The AT&T compiler was originally chosen because it was free (to Universities) and because we thought it would be easier to modify an existing compiler than building one from scratch. We decided to build a new compiler rather than continue to use the AT&T compiler for three reasons. First, the current version of C++ is 2.1, not 1.2. It would have been very difficult to modify the 1.2 compiler to accept 2.1 code. Further, continued modification of their existing baroque code was becoming more and more difficult. Second, proprietary reasons prevented us from freely distribut-

ing our compiler as long as it was based upon the AT&T code. Third, making language changes, such as those envisioned to support Real-Time Mentat, is easier if the grammar and production rules are designed from the beginning to facilitate language experimentation (as ours is).

The compiler accepts a subset of the full MPL/A. In particular it does not accept select/accept statements, priorities, or permit inheritance for Mentat classes. The compiler has been distributed to over sixty sites world-wide, and is in active use at five sites in the United States. Further, the lexer and parser are being used by researchers at the University of Illinois. The compiler is fairly robust. However, there are several known bugs. These are detailed in TR-91-31 and TR-91-32.

4.0 Application Results

Part of our research plan for the Summer of 1991 was to implement two applications using Mentat/A and to collect performance results. The two applications selected in consultation with the Information Processing Technical Branch were the NASA graph 7, and the Lunar docking application from JPL. Below we provide pseudo-code for each and the performance both with and without the ATAMM firing rules enforced. We have also implemented a pipeline application. We added this application because it most clearly illustrates the differences between Mentat/A and regular Mentat.

Only the initial ATAMM rules were implemented and tested. These rules have the effect of limiting performance is some circumstances. The results do not necessarily apply to the enhanced ATAMM. Further, ATAMM analysis was not performed to determine the appropriate injection rate or queue depths for the given number of processors for any of the problems below.

4.1 Pipe - A Pipeline Program

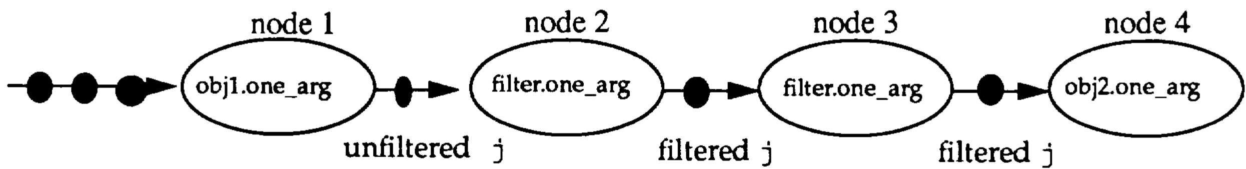
The pipeline program most clearly illustrates both pipeline parallelism in Mentat/A and the effect of the ATAMM rules, in particular the queue depth rules. The code fragment for the

main loop is shown below in Figure 4-(a). The corresponding graph is shown in Figure 4-(b).

```
persistent mentat class generic{        regular mentat class gfilter{
public:                                  public:
   int one_arg(int arg1);                   int one_arg(int arg1);
}                                        }

generic obj1, obj2;
gfilter filter;
obj1.create();
obj2.create();
int i;
 for (i=0; i < MAX_ITERATIONS; i++) {
   int j;
   j = obj1.one_arg(delay);
   j = filter.one_arg(j);
   j = filter.one_arg(j);
   j = obj2.one_arg(j);
}
```

(a) Pipeline code fragment.



(b) Execution graph for pipeline.

Figure 4. Pipeline program and execution graph.

Note that there are no self loops in the code. The performance results when the time required to complete nodes 2 & 3 is varied from 0 to 4 time units, and the time to execute nodes 1 & 4 is held at 1 time unit, are shown in Table 1. Twenty iterations of the loop were performed.

Table 1: ATAMM Pipe with 8 processors

Time for 2&3 (Seconds)	TBO - Q1 mSec	TBIO - Q1 mSec
0	1170	3040
1	1500	6670
2	2500	11150
3	3253	14900
4	4269	18450

Suppose now we had four processors instead of eight. For this experiment we varied the queue depth of nodes 2 & 3 from one to three. We would expect TBIO to be worse as queue depth increases, and unchanged for a queue depth of one. This is confirmed in Table 2. This can be contrasted with the pure Mentat performance where the injection rate is unconstrained by queue rules. Note that while TBO remains good, TBIO increases dramatically. In particular compare the Mentat TBIO-4 column with the ATAMM TBIO-Q1 column. In Table 3 20 iterations of the graph were used. We found that as the number of iterations used was increased, TBO remained constant, but that TBIO increased dramatically. This was not the case when the ATAMM rules were used, clearly demonstrating that the inclusion of the ATAMM rules provides better control.

Table 2: ATAMM Pipe with 4 processors

Time for 2&3 (Seconds)	TBO - Q1 mSec	TBIO -Q1 mSec	TBO-Q2 mSec	TBIO-Q2 mSec	TBO-Q3 mSec	TBIO-Q3 mSec
0	1250	3150	1330	2840	1360	3043
1	1670	6440	1960	9650	1870	8710
2	2520	11980	2010	15390	1930	16670
3	4870	21100	2790	24390	2460	24250
4	5150	21320	3830	35460	4330	43290

Table 3: Unconstrained Mentat performance

Time for 2&3 (Seconds)	TBO-4 mSec	TBIO-4 mSec	TBO-8 mSec	TBIO-8 mSec
0	1180	2710	1120	2840
1	1580	7080	1430	7610
2	2510	19030	2400	21660
3	3460	30980	3550	33130
4	5110	53010	4410	43660

4.2 NASA Graph 7

NASA graph 7 (Figure 5) is an example from IPTB that we have implemented in MPL/A. In this example we can control the injection rate using a `nasa_source` object. Thus we can determine if limiting the injection rate has the same effect as modifying the queue depths. In the

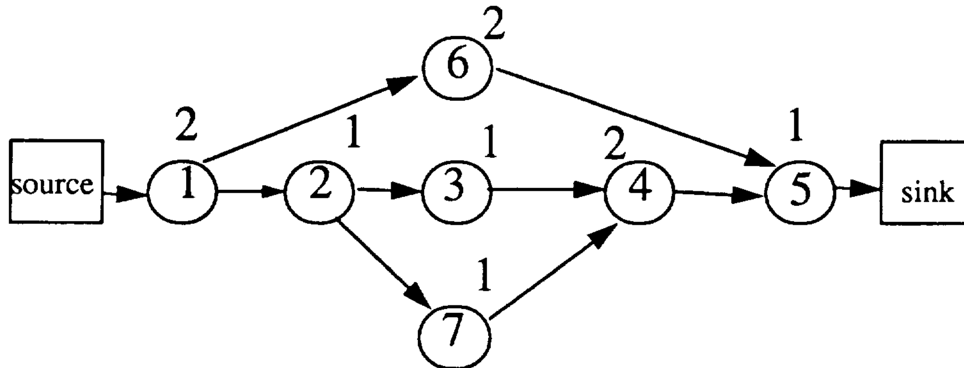


Figure 5. NASA graph 7. Node computation times appear above the nodes.

experiments the injection interval is how often we *attempted* to inject tokens, e.g., every 0, 1, or 2 time units. The actual rate may be lower due to the enforcement of ATAMM firing rules. This is

Table 4: Graph 7 - Mentat/A implementation

injection interval (Seconds)	TBO - Q1 mSec	TBIO -Q1 mSec	TBO -Q2 mSec	TBIO -Q2 mSec
0	4520	15430	3705	16220
1	4590	15230	3880	16220
2	4270	12810	3730	13890
3	4670	13590	3620	13300
4	4240	10760	4080	9811
5	5065	9910	4850	8620
6	5770	8150	5800	8010
7	6750	8130	6670	7921
8	7660	7685	7530	7790

the case, for example, when the injection interval is zero. Data were collected for one to eight pro-

cessors. We have shown data for the four processor case. The effect of the firing rules, and of limiting the injection rate can be seen when the data in Table 4 is compared to the unconstrained Mentat implementation data shown in Table 5. Note in particular how Mentat/A preserves TBIO

Table 5: Graph 7 - Regular Mentat implementation.

injection interval (Seconds)	TBO - Q1 mSec	TBIO -Q1 mSec
0	1010	30520
1	1230	26390
2	2580	20650
3	3130	14520
4	3350	12560
5	4780	9470
6	5460	9160
7	6360	7880
8	7180	7590

with the current queue settings, while the regular Mentat preserves TBO (throughput) at the expense of TBIO. Keep in mind that only ten iterations were performed. If more iterations were performed the regular Mentat version TBIO would have been worse as additional tokens were injected into an already saturated system. Note also how controlling the injection rate in the pure Mentat version has much the same effect as enforcing the ATAMM rules.

4.3 JPL Lunar Docking System

The JPL Lunar docking program (hereafter the docker) was originally written for project MAX. The code is written in an extended C and consists of thirteen modules. These modules perform functions such as propulsion management and orbital dynamics calculations. There is one characteristic that all of the modules share, small granularity. The orbital dynamics model code is one of the larger computationally. It consists of one conditional, nine assignments, and twenty

floating point operations. This is far too small-grained for most parallel processing systems to successfully exploit. Overhead will dominate, causing worse performance than could be realized with a sequential implementation.

We translated the docker from C to MPL/A in the late summer and early fall of 1991. The structure of the MPL/A program preserves as much as possible of the original program. (A copy of the MPL/A code is included in Appendix B.) We are interested in the maximum sustainable injection rate, i.e., how fast can we iterate the graph. The presence of cycles in the graph constrains parallelism. Below we contrast the Mentat/A with the regular Mentat times. All times are

Table 6: JPL Lunar Docker Results

injection interval	ATAMM iterations	ATAMM rate	Mentat iterations	Mentat rate
50mS	350	170mS	NA	NA
60mS	390	153mS	NA	NA
70mS	392	153mS	747	80mS
80mS	361	168mS	747	80mS
90mS	369	162mS	598	100mS
100 mS	405	148mS	597	100mS
500 mS	120	500mS	120	500mS
1000 mS	60	1000mS	60	1000mS
2000 mS	30	2000mS	30	2000mS

in milli-seconds. To determine the sustainable iteration rate we ran the application for sixty seconds and counted the actual number of iterations executed. We then divide the total time by number of iterations, obtaining iteration rate. For example, 120 iterations yields a rate of 500mS.

There are two things to note in the data. First, the maximum iteration rate for the ATAMM version is in the range 150mS-170mS per iteration. The enforcement of the ATAMM rules prevents a higher injection rate because of the overhead of our implementation. Second, the pure Mentat version suffers from buffer overflows when the injection interval is less than 70mS. Thus, while the Mentat version can sustain much higher rates, i.e. every 80mS, it lacks mechanism to

prevent overflow. This is the clear advantage of the ATAMM approach, it prevents overflow.

4.4 Observations

A few observations are in order before we proceed. First, the effect of unconstrained parallelism, as in the pure Mentat case, is that good CPU utilization, low TBO's, and high TBIO's are realized. Thus, TBIO is traded off for TBO by default. Second, the Mentat/A implementation successfully constrained parallelism by implementing the ATAMM firing rules. Further, by manipulating queue depths we can trade TBO for TBIO. Third, the overhead of enforcing the ATAMM firing rules is not that large. This can be seen by comparing the Mentat/A results of Table 2 with the pure Mentat results of Table 4. However, we did find that our implementation was flawed in such a manner that the performance would suffer when actor times were large. The problem is that while messages are accepted in an interrupt driven fashion, they are not acted upon until an object checks for the next request. Thus, ATAMM_QUERY's may not be serviced for some time even if there is an empty queue slot. This delays the ancestor, which in turn delays his ancestor, and so on.

5.0 Summary

We have successfully completed our objectives. We built and tested under a variety of applications Mentat/A. The results clearly show the effect of enforcing the initial ATAMM firing rules. The MPL compiler was constructed. It generates code to automatically generate data flow program graphs. The compiler has been distributed to over forty-five sites world-wide, and is in regular use at five.

We also learned that a good implementation of ATAMM with Mentat was harder than we at first believed. The primary problem was not with rule enforcement, but rather with where to enforce the rules. A good implementation of ATAMM requires that the rules be enforced in a distributed run-time system, not in objects as we did. In particular, Mentat's assumption about an initial locus of control does not mesh well with ATAMM. Finally we learned that in some cases the initial ATAMM rules are superfluous, that controlling injection rate is sufficient to control the

TBO/TBIO trade-off. This is not a general result though, there is a large class of problems for which controlling the injection rate is insufficient.

Finally, our implementation results apply only to the initial ATAMM, not to the enhanced ATAMM. The enhanced ATAMM design has better mechanism for increasing and controlling asynchrony between nodes, improving performance while maintaining control over the TBO/TBIO trade-off.

Appendix A: Supported Papers and Conference Trips

Grimshaw, Andrew S., and Virgilio E. Vivas, "FALCON: A Distributed Scheduler for MIMD Architectures", *Proceedings of the Symposium on Experiences with Distributed and Multiprocessor Systems*, Atlanta, GA, March, 1991.

Grimshaw, Andrew S., and Mriganka Das, "High Performance Parallel File Objects", *Proceedings of the Sixth Distributed Memory Computing Conference*, Portland, OR., April 29-May 1, 1991.

Grimshaw, Andrew S.,"Meta-Systems: An Approach Combining Parallel Processing and Heterogeneous Distributed Computing Systems", Proceedings Sixth International Parallel Processing Symposium Workshop on Heterogeneous Processing, Beverly Hills, CA, March 23-26.

Grimshaw, Andrew S., Edmond C. Loyot Jr., and Jon B. Weissman, "Mentat Programming Language (MPL) Reference Manual", Computer Science TR-91-31, University of Virginia, November, 1991.

Grimshaw, Andrew S., et al, "Mentat Users Manual", Computer Science TR-91-32, University of Virginia, November, 1991.

Attended 11th Symposium on Real-Time Systems, Dec. 4-7, 1990, Orlando, Florida.

Attended Sixth Distributed Memory Computing Conference, Portland, OR., April 29-May 1, 1991.

Appendix B: Lunar Docker

```
// File:  classdefs.h
/* All of the Mentat class definitions are here.          */

#include "typedefs.h"

persistent mentat class thruster_model
{
        int x;
public:
        ThrusterOutputToken thruster(ThrusterStateToken);
};


persistent mentat class orb_dyn_model
{
        int x;
        OrbitStateToken const_state;
public:
        OrbitStateToken orb_dyn(ThrusterOutputToken);
        int init(OrbitStateToken);
};


persistent mentat class radar_alt_model
{
        int x;
public:
        AltitudeToken radar_alt(OrbitStateToken);
};


persistent mentat class radio_range_model
{
        int x;
public:
        RangeToken radio_range(OrbitStateToken);
};


persistent mentat class accel_estimator
{
        int x;
public:
        AccelerationToken accel(ThrusterStateToken);
};


persistent mentat class orbit_dynamics
{
        OrbitStateToken const_state;
public:
        OrbitStateToken orbit(CorrectionToken, AccelerationToken);
        int init(OrbitStateToken);
};


persistent mentat class data_output
{
        LongToken data_count;
public:
        void data_out(LongReportToken);
};
```

```
persistent mentat class corrector_model
{
        data_output data_o;
public:
        CorrectionToken corrector(OrbitStateToken, RangeToken, AltitudeToken);
        int cor_init(data_output);
};


persistent mentat class controller_model
{
        ControlStateToken control_state;
        data_output data_o;
public:
        ThrusterStateToken controller(OrbitStateToken);
        void control_law(OrbitStateToken, ThrusterStateToken);
        int con_init(data_output);
};


persistent mentat class propulsion_man {
        thruster_model thruster_m;
        accel_estimator accel_e;
        orb_dyn_model orb_dyn_m;
        radar_alt_model radar_alt_m;
        radio_range_model radio_range_m;
        orbit_dynamics orbit_d;
        corrector_model corrector_m;
        controller_model controller_m;
        data_output data_o;
        int first;
        ThrusterStateToken propul_state;
        CorrectionToken cor;
        int iterations;
        int shut;
public:
        int propul_init(data_output);
        void update(ThrusterStateToken);
        void iterate(ThrusterStateToken);
        int shutdown(int);
};


persistent mentat class scheduling
{
        TimeToken time;
        propulsion_man manager;
        data_output data_o;
public:
        ThrusterStateToken schedul(int, int);
        int sch_init(propulsion_man, data_output);
};
```

```
//      File:  typedefs.h
#ifndef _typedefs
#define _typedefs

//      DOCKING DEMO DEFINITIONS

#define pi      (3.14159265358979323846)

//(* ((float*)(&bugA1)) ) sec
#define deltaT  (0.25)

//radians per sec
#define omega   (2.0 * pi / 256.0)

//meters
#define rMoon   (648.7330863)

//meters
#define rStation        (51200.0 / (2 * pi))

//m per sec squared
#define thrust  (1)

//m per sec squared
#define sigmaT  (0.05 * thrust)

//meters
#define sigmaV  (1.0)

//These "bug"s are defined by SAS
#define bugA2 (1000)

#define bugA3 (1000)

#define bugA4 (100)

#define bugl4 (100)


//      TOKEN TYPEDEFS

#ifndef FALSE
#define FALSE (0)
#define TRUE (!FALSE)
#endif

typedef int     Boolean;

#define upward  0
#define downward 1
#define forward 2
#define backward 3

typedef int     Direction;

struct OrbitStateToken
{
        float   hPosition;
        float   vPosition;
        float   hVelocity;
        float   vVelocity;
};
```

```
#define DONTCARE        2
#define OFF             0
#define ON              1

typedef int             TriState;

typedef enum { ThrusterCommand, ThrusterEnable } ThrusterStateType;

struct ThrusterStateToken
{
        ThrusterStateType       type4;
        TriState                thrusterStateUpward;
        TriState                thrusterStateDownward;
        TriState                thrusterStateForward;
        TriState                thrusterStateBackward;
};

struct  ReportToken
{
        char report[32];
};

struct LongReportToken
{
        char LongReport[120];
};

struct RangeToken
{
        float range;
};

struct AltitudeToken
{
        float altitude;
};

struct AccelerationToken
{
        float acc_h;
        float acc_v;
};

struct CorrectionToken
{
        float cor_h;
        float cor_v;
        float sh;
        float sv;
};

typedef enum { OK, Dock, Damage, Impact } StatusMessage;

struct ControlStateToken
{
        Boolean zoom;
        StatusMessage message;
};

struct TimeToken
{
        long time;
};
```

```
struct LongToken
{
        long longtoken;
};

struct ThrusterOutputToken
{
        CorrectionToken corr_zero;
        AccelerationToken accel;
};
#endif
```

```
#include "typedefs.h"
#include "classdefs.h"
#include <stdio.h>

AccelerationToken accel_estimator::accel(ThrusterStateToken actuation)
{
        AccelerationToken est_accel;
#ifdef DEBUG
        printf("called accel_estimator\n");
        fflush(stdout);
#endif

        est_accel.acc_h = 0.0;

        if (actuation.thrusterStateForward == ON)
          est_accel.acc_h += thrust;

        if (actuation.thrusterStateBackward == ON)
          est_accel.acc_h -= thrust;

        est_accel.acc_v = 0.0;

        if (actuation.thrusterStateUpward == ON)
          est_accel.acc_v += thrust;

        if (actuation.thrusterStateDownward == ON)
          est_accel.acc_v -= thrust;

        rtf(est_accel);
        return(est_accel);
}
```

```
#include "typedefs.h"
#include "classdefs.h"
#include <stdio.h>
#include <math.h>

int controller_model::con_init(data_output dout)
{
        data_o = dout;
        rtf(0);
}


ThrusterStateToken controller_model::controller(OrbitStateToken est_state)
{
#define hP est_state.hPosition
#define vP est_state.vPosition
#define hV est_state.hVelocity
#define vV est_state.vVelocity

        ThrusterStateToken thr_cmd2;
        char zoomReportBuffer[64], *fromPtr, *toPtr, c;

#ifdef DEBUG
        printf("called controller\n");
        fflush(stdout);
#endif


        zoomReportBuffer[0] = 0;

        if ((hP <= 0.0) && (hP >= -1.0) && (vP <= 2.0) && (vP >= -2.0))
          {
            if (control_state.message != Dock)
              {
                sprintf((char *)zoomReportBuffer, "!SDocking successful!\n");
                control_state.message = Dock;
              }
          }
        else if ((!control_state.zoom) && (hP < 280.0) && (hP > -260.0) &&
                 (vP < 180.0) && (vP > -180.0))
          {
            control_state.zoom = TRUE;
            sprintf((char *)zoomReportBuffer, "!Z1\n");
          }
        else if (control_state.zoom && ((hP > 300.0) || (hP < -280.0) ||
                                        (vP > 200.0) || (vP < -200.0)))
          {
            control_state.zoom = FALSE;
            sprintf((char *)zoomReportBuffer, "!Z0\n");
          }
        else if (control_state.zoom && (((hP > 6.0) && (hP <= 92.0) &&
                 (vP <= 22.0) && (vP >= -22.0)) || ((hP <= 71.0) &&
                 (((hP >= 30.0) && (vP <= 58.0) && (vP >= -58.0)) ||
                  ((hP >= 14.0) && (vP <= 30.0) && (vP >= -30.0))))))
          {
            if (control_state.message != Damage)
              {
                sprintf((char *)zoomReportBuffer, "!SCrunch!! Ship damaged!\n");
                control_state.message = Damage;
              }
          }
        else if (vP < -(rStation - rMoon))
          {
            if (control_state.message != Impact)
```

```
              {
                sprintf((char *)zoomReportBuffer, "!SCRASH!!! Surface impact!!\n");
                control_state.message = Impact;
              }
          }
        else if (control_state.message != OK)
          {
            sprintf((char *)zoomReportBuffer, "!S \n");
            control_state.message = OK;
          }

        if (zoomReportBuffer[0] != 0)
          {
            LongReportToken status_report;

            fromPtr = zoomReportBuffer;
            toPtr = status_report.LongReport;

            do { *toPtr++ = (c = *fromPtr++);} while (c != 0);
            //data_o.data_out(status_report);
          }

        rtf(thr_cmd2);
        return(thr_cmd2);
}


void controller_model::control_law(OrbitStateToken est_state, ThrusterStateToken thr_cmd2)
{
#define GAIN1 0.1
#define GAIN2 20.0
#define GAIN3 160.0

       float errorH, errorV, vAvg, vBias;

#ifdef DEBUG
       printf("called control_law\n");
        fflush(stdout);
#endif

        if (hP < -300.0)
          {
            vBias = hP / 1000.0;

            if (vBias > 3.0)
              vBias = 3.0;
            else if (vBias < -3.0)
              vBias = -3.0;

            vAvg = ((hP < -0.1) ? ((-1.0 + vBias) / omega) :
                    ((1.0 + vBias) / omega));

            errorH = (GAIN1) * ((hV / 2.0) - omega * ((vAvg / 4.0) - vP));

            if ((fabs(errorH) < 1) && (fabs(vV) < 1.0) && ((vBias * hV) > -50.0))
              errorH = vBias;

            errorV = omega * (2.0 * (GAIN2) * vV + omega * (GAIN3) *
                            (vP - vAvg));
          }
        else
          {
            errorH = (hP < -0.1) ? 4.0 * (hV - 0.5 + hP / 40.0) : 0.0;
            errorV = vP + 5.0 * vV;
```

```
            }

         if (errorH >= 2)
            {
              thr_cmd2.thrusterStateForward = OFF;
              thr_cmd2.thrusterStateBackward = ON;
            }
         else if (errorH <= -2)
            {
              thr_cmd2.thrusterStateBackward = OFF;
              thr_cmd2.thrusterStateForward = ON;
            }
         else
            {
              thr_cmd2.thrusterStateForward = OFF;
              thr_cmd2.thrusterStateBackward = OFF;
            }

         if (errorV >= 2)
            {
              thr_cmd2.thrusterStateUpward = OFF;
              thr_cmd2.thrusterStateDownward = ON;
            }
         else if (errorH <= -2)
            {
              thr_cmd2.thrusterStateUpward = ON;
              thr_cmd2.thrusterStateDownward = OFF;
            }
         else
            {
              thr_cmd2.thrusterStateUpward = OFF;
              thr_cmd2.thrusterStateDownward = OFF;
            }

         thr_cmd2.type4 = ThrusterCommand;

        rtf(0);
}
```

```
#include "typedefs.h"
#include "classdefs.h"
#include <stdio.h>
#include <math.h>

int corrector_model::cor_init(data_output dout)
{
        data_o = dout;
        rtf(0);
}


CorrectionToken corrector_model::corrector(OrbitStateToken est_state, RangeToken meas_rng,
 AltitudeToken meas_alt)
{
        CorrectionToken corr;
        LongReportToken state_report;

        float absHPosition, vPosn, ePlus, eMinus, hError, vError, sigmaHInverse,
              root_sigmaHInverse, kGainH, kGainSH, kGainV, kGainSV, temp;

        vPosn = meas_alt.altitude - (rStation - rMoon);

        vError = vPosn - est_state.vPosition;

        temp = pow(meas_rng.range, 2) - pow(vPosn, 2);

        if (temp > 0)
          absHPosition = sqrt(temp);
        else
          absHPosition = 0;

        ePlus = absHPosition - est_state.hPosition;
        eMinus = -absHPosition - est_state.hPosition;

        if (fabs(ePlus) <= fabs(eMinus))
          hError = ePlus;
        else
          hError = eMinus;

        temp = meas_rng.range / vPosn;

        if (temp > 1.0)
          sigmaHInverse = 1.0 - (2.0 / (pow(temp, 2) + 1.0));
        else
          sigmaHInverse = 0;

        kGainSH = sigmaT * sigmaHInverse;
        kGainH = sqrt(2.0 * kGainSH);
        kGainSV = sigmaT / sigmaV;
        kGainV = sqrt(2.0 * kGainSV);

        corr.cor_h = kGainH * hError;
        corr.cor_v = kGainV * vError;
        corr.sh = kGainSH * hError;
        corr.sv = kGainSV * vError;

        sprintf((char *)state_report.LongReport, "!P%f,%f\n!V%f,%f\n!R%f\n!A%f\n",
                est_state.hPosition, est_state.vPosition, est_state.hVelocity,
                est_state.vVelocity, meas_rng.range, meas_alt.altitude);
        //data_o.data_out(state_report);

#ifdef DEBUG
        printf("called corrector\n");
```

```
        fflush(stdout);
#endif
        rtf(corr);
        return(corr);
}
```

```
#include "typedefs.h"
#include "classdefs.h"
#include<stdio.h>

void data_output::data_out(LongReportToken merge_report)
{
#ifdef DEBUG
        printf("called data_output\n");
        fflush(stdout);
#endif
        FILE *outfile=fopen("output","a+");

        data_count.longtoken += 1;

        fprintf (outfile,"%s", merge_report.LongReport);
        fflush(outfile);
        fclose(outfile);

        rtf(0);
}
```

```
/*      The main program calls reads command line arguments,
        initializes the propulsion_manager, scheduler, and output.
        It then starts everything going and waits for completion
*/
#include <stdio.h>
#include "classdefs.h"
#include <libc.h>

int main(int argc, char **argv)
{
        ThrusterStateToken c1;
        ThrusterStateToken c2;
        propulsion_man pm;
        scheduling sch;
        data_output dout;
        int i,j,total_time, interval;
        int iterations;

        if (argc == 3)
          {
            //Check for the arguments, total_time and interval
            total_time = atoi(argv[1]);
            interval = atoi(argv[2]);
          }
        else
          {
            //else use defaults
            total_time = 3000;
            interval = 1000;
          }

        printf("Lunar docker, total_time %d, interval %d\n",
                        total_time,interval);
        fflush(stdout);
        // Create the propulsion manager, scheduler, and data_output
        pm.create();
        sch.create();
        dout.create();
        // Now initialzie them.
        i = pm.propul_init(dout);
        i=sch.sch_init(pm, dout);

        // schedul returns when all iterations have been done.
        c2=sch.schedul(total_time, interval);
        // So, block and wait for completion.
        c1=c2;
        // Now shut everything down.
        iterations = pm.shutdown(0);
        // shutdown tells us how many iterations were performed.
        // it also kills off all of its children.
        printf("actual iterations = %d\n",iterations);
        fflush(stdout);
        // Now destroy the objects.
        pm.destroy();
        sch.destroy();
        dout.destroy();
}
```

```
#include "typedefs.h"
#include "classdefs.h"
#include <stdio.h>
#include <math.h>

OrbitStateToken orb_dyn_model::orb_dyn(ThrusterOutputToken thruster_res)
{
        OrbitStateToken new_state;
        float twoOmega;

#ifdef DEBUG
        printf("called orb_dyn_model\n");
        fflush(stdout);
#endif

        //left out if "(!bug11 &&"
        if ((const_state.hPosition <= 0.0) && (const_state.hPosition >= -1.0) &&
            (const_state.vPosition <= 2.0) && (const_state.vPosition >= -2.0))
        {
          const_state.hPosition = -0.04;
          const_state.vPosition = 0;
          const_state.hVelocity = 0;
          const_state.vVelocity = 0;
        }
        else
        {
          twoOmega = 2.0 * omega;

          const_state.hPosition = const_state.hPosition +
            (const_state.hVelocity + thruster_res.corr_zero.cor_h) * deltaT;

          const_state.vPosition = const_state.vPosition +
            (const_state.vVelocity + thruster_res.corr_zero.cor_v) * deltaT;

          const_state.hVelocity = const_state.hVelocity +
            (-(twoOmega * const_state.vVelocity) +
            thruster_res.accel.acc_h + thruster_res.corr_zero.sh) * deltaT;

          const_state.vVelocity = const_state.vVelocity +
            ((3.0 * pow(omega, 2)) * const_state.vPosition +
             twoOmega * const_state.hVelocity +
             thruster_res.accel.acc_v + thruster_res.corr_zero.sv) * deltaT;
        }

        rtf(new_state);
        return(new_state);
}

int orb_dyn_model::init(OrbitStateToken merge_state)
{
        printf("called orb_dyn_model::init\n");
        fflush(stdout);

        const_state = merge_state;
        rtf(0);
}
```

```
#include "typedefs.h"
#include "classdefs.h"
#include <stdio.h>

OrbitStateToken orbit_dynamics::orbit(CorrectionToken corr, AccelerationToken accel)
{
//variable names picked by SAS
        OrbitStateToken est_state;

#ifdef DEBUG
        printf("called orbit_dynamics\n");
        fflush(stdout);
#endif

        rtf(est_state);
        return(est_state);
}

int orbit_dynamics::init(OrbitStateToken merge_state)
{
#ifdef DEBUG
        printf("called orbit_dynamics::init\n");
        fflush(stdout);
#endif

        const_state = merge_state;
        rtf(0);
}
```

```
#include "typedefs.h"
#include "classdefs.h"
#include <stdio.h>
int total_iterations;
#ifdef ATAMM
// used for debugging
extern int atamm_threshold;
#endif

int propulsion_man::shutdown(int x)
{
#ifdef DEBUG
        printf("in prop man shutdown\n");
        fflush(stdout);
#endif
        shut= TRUE;
        if (iterations ==0) {
                thruster_m.destroy();
                accel_e.destroy();
                orb_dyn_m.destroy();
                radar_alt_m.destroy();
                radio_range_m.destroy();
                orbit_d.destroy();
                corrector_m.destroy();
                controller_m.destroy();
#ifdef DEBUG
                printf("exiting prop man shutdown\n");
                fflush(stdout);
#endif
                rtf(total_iterations);
        }
        else {
#ifdef DEBUG
                printf("pending shutdown\n");
                fflush(stdout);
#endif
        }
        printf("total iterations = %d\n",total_iterations);
        return(0);
}

int propulsion_man::propul_init(data_output dout)
{
        int i,j;
#ifdef DEBUG
        printf("in prop man init\n");
        fflush(stdout);
#endif
#ifdef ATAMM
        printf("atamm threshold = %d\n",atamm_threshold);
#endif
        OrbitStateToken init_state;
        data_o = dout;

        // First create children.
        thruster_m.create();
        accel_e.create();
        orb_dyn_m.create();
        radar_alt_m.create();
        radio_range_m.create();
        orbit_d.create();
        corrector_m.create();
        controller_m.create();
```

```
        first = TRUE;

        init_state.hPosition = bugA2;
        init_state.vPosition = bugA3;
        init_state.hVelocity = bugA4;
        init_state.vVelocity = bug14;

        i=orb_dyn_m.init(init_state);
        i=orbit_d.init(init_state);
        i=corrector_m.cor_init(data_o);
        i=controller_m.con_init(data_o);
        j=i;

        rtf(0);
        iterations =0;
        total_iterations =0;
        shut= FALSE;
        // Initialize correction
        cor.cor_h=0.0;
        cor.cor_v=0.0;
        cor.sh=0.0;
        cor.sv=0.0;
        // initialize propulsion state.
        propul_state.thrusterStateUpward =0;
        propul_state.thrusterStateDownward =0;
        propul_state.thrusterStateForward =0;
        propul_state.thrusterStateBackward =0;
        propul_state.type4 =ThrusterEnable;
#ifdef DEBUG
        printf("in prop man done\n");
        fflush(stdout);
#endif
        return 0;
}

void propulsion_man::iterate(ThrusterStateToken merge_thr_cmd)
{
        ThrusterStateToken actuation;
        {
        LongReportToken actuation_report;

#ifdef DEBUG
        printf("called propul_man::iterate\n");
        fflush(stdout);
#endif

        ThrusterOutputToken tm;
        AccelerationToken ae;
        OrbitStateToken odm;
        AltitudeToken ram;
        RangeToken rrm;
        OrbitStateToken od;
        ThrusterStateToken con;
        CorrectionToken new_cor;
        new_cor = cor;
        actuation = propul_state;

        actuation.type4 = ThrusterEnable;

        tm = thruster_m.thruster (actuation);
        ae = accel_e.accel (actuation);

        odm = orb_dyn_m.orb_dyn (tm);
```

```
        ram = radar_alt_m.radar_alt (odm);
        rrm = radio_range_m.radio_range (odm);

        od = orbit_d.orbit (cor, ae);

        cor = corrector_m.corrector (od, rrm, ram);
        con = controller_m.controller (od);

        SELF.update(con);
        iterations++;
        total_iterations ++;
        first = FALSE;

        sprintf((char *)actuation_report.LongReport, "!T%d,%d\n!T%d,%d\n!T%d,%d\n!T%d,%d\n
",
                upward, actuation.thrusterStateUpward,
                downward, actuation.thrusterStateDownward,
                forward, actuation.thrusterStateForward,
                backward, actuation.thrusterStateBackward);

        data_o.data_out(actuation_report);
        // Note: data out not called for timing tests, I/O dominated.

#ifdef DEBUG
        printf("at end of propul_man::iterate\n");
        fflush(stdout);
#endif
        }
        rtf(0);
#ifdef DEBUG
        printf("have done rtf at  end of propul_man::iterate\n");
        fflush(stdout);
#endif
        return;
}


void propulsion_man::update(ThrusterStateToken merge_thr_cmd)
{
#ifdef DEBUG
          printf("called propul_man::update\n");
        fflush(stdout);
#endif
        iterations--;

          propul_state.type4 = ThrusterCommand;

          if (merge_thr_cmd.thrusterStateUpward != DONTCARE)
            propul_state.thrusterStateUpward = merge_thr_cmd.thrusterStateUpward;

          if (merge_thr_cmd.thrusterStateDownward != DONTCARE)
            propul_state.thrusterStateDownward =
              merge_thr_cmd.thrusterStateDownward;

          if (merge_thr_cmd.thrusterStateForward != DONTCARE)
            propul_state.thrusterStateForward =
              merge_thr_cmd.thrusterStateForward;

          if (merge_thr_cmd.thrusterStateBackward != DONTCARE)
            propul_state.thrusterStateBackward =
              merge_thr_cmd.thrusterStateBackward;

          rtf(0);
        if ((iterations==0)&&(shut==TRUE)) {
```

```
            rtf(SELF.shutdown(1));
        }
}
```

```
#include "typedefs.h"
#include "classdefs.h"
#include <stdio.h>

AltitudeToken radar_alt_model::radar_alt(OrbitStateToken state)
{
        AltitudeToken meas_alt;
        float measurementNoise;

#ifdef DEBUG
        printf("called radar_alt_model\n");
        fflush(stdout);
#endif

        measurementNoise = 0.0;
        meas_alt.altitude = state.vPosition + (rStation - rMoon) +
          measurementNoise;

        rtf(meas_alt);
        return(meas_alt);
}
```

```
#include "typedefs.h"
#include "classdefs.h"
#include <stdio.h>
#include <math.h>

RangeToken radio_range_model::radio_range(OrbitStateToken state)
{
        RangeToken meas_rng;
        float measurementNoise;

#ifdef DEBUG
        printf("called radio_range_model\n");
        fflush(stdout);
#endif

        measurementNoise = 0.0;

        meas_rng.range = sqrt(pow(state.hPosition, 2) +
                              pow(state.vPosition, 2)) + measurementNoise;

        rtf(meas_rng);
        return(meas_rng);
}
```

```
#include "typedefs.h"
#include "classdefs.h"
#include <stdio.h>

int scheduling::sch_init(propulsion_man po, data_output dout)
{
//Initialize members, manager and data_o
        printf("in sch init\n");
        fflush(stdout);
        manager=po;
        data_o = dout;
        rtf(0);
}

ThrusterStateToken scheduling::schedul(int total_time, int interval)
{
//Control the execution of the graph.  Graph will execute every interval
//iterations.  It will execute total_time/interval times.  Each time
//call manager.iterate and generate a time_report.

        ThrusterStateToken thr_cmd0;
        LongReportToken time_report;
        mentat_timer temp;
        int first, num_times;
        unsigned start_t;

#ifdef DEBUG
        printf("called scheduling\n");
        fflush(stdout);
#endif

        //initialize thr_cmd0
        thr_cmd0.type4 = ThrusterEnable;
        thr_cmd0.thrusterStateUpward = DONTCARE;
        thr_cmd0.thrusterStateDownward = DONTCARE;
        thr_cmd0.thrusterStateForward = DONTCARE;
        thr_cmd0.thrusterStateBackward = DONTCARE;

        first = TRUE;

        num_times = total_time / interval;
        start_t = temp.get_time();

        for (int i=0; i<num_times; i++)
          {
            //delay for "interval"
            unsigned time_val = temp.get_time();
            if (time_val > start_t + total_time) break;
            int secs = interval / 1000;
            sleep (secs);
            while ((temp.get_time() - time_val) < interval);

            //call propulsion_man::iterate
            manager.iterate(thr_cmd0);

            //Create a time_report by calling data_output
            if (first == TRUE)
              {
                sprintf((char *)time_report.LongReport, "!S \n!E%d\n",
                        (long)((time.time) / 100));
                first = FALSE;
              }
            else
              {
```

```
                sprintf((char *)time_report.LongReport, "!E%d\n",
                        (long)((time.time) / 100));
            }
            data_o.data_out(time_report);
            // Note: data out not called in timing tests.
          }

        rtf(thr_cmd0);
        return(thr_cmd0);
}
```

```
#include "typedefs.h"
#include "classdefs.h"
#include <stdio.h>

ThrusterOutputToken thruster_model::thruster(ThrusterStateToken actuation)
{
        ThrusterOutputToken     thrust_comp;
        float thrusterNoise;

#ifdef DEBUG
        printf ("called thruster_model\n");
        fflush(stdout);
#endif

        thrusterNoise = 0.0;

        thrust_comp.accel.acc_h = thrusterNoise;

        if (actuation.thrusterStateForward == ON)
          thrust_comp.accel.acc_h += thrust;

        if (actuation.thrusterStateBackward == ON)
          thrust_comp.accel.acc_h -= thrust;

        thrust_comp.accel.acc_v = thrusterNoise;

        if (actuation.thrusterStateUpward == ON)
          thrust_comp.accel.acc_v += thrust;

        if (actuation.thrusterStateDownward == ON)
          thrust_comp.accel.acc_v -= thrust;

        thrust_comp.corr_zero.cor_h = thrust_comp.corr_zero.cor_v =
        thrust_comp.corr_zero.sh = thrust_comp.corr_zero.sv = 0;

        rtf(thrust_comp);
        return(thrust_comp);
}
```

"Pipe"

```
#include <stdio.h>
#include <osfcn.h>
#include "ggeneric_mc.h"
#include "nasa_sink.h"

extern mentat_object SELF;

int main(int argc,char **argv) {
    int iterations, delay, fdelay;
    iterations = 10;
    delay = 0;
    fdelay = 0;
    if (argc!=4) {
        printf("usage: mpipe delay filter_delay iterations\n");
        exit(0);
    }
    delay = atoi(argv[1]);
    delay*=1000;
    fdelay = atoi(argv[2]);
    fdelay*=1000;
    iterations = atoi(argv[3]);
    int temp,temp2,i;
    mentat_timer interval;
    nasa_statistic stats;
    ggeneric_mc node1, node2;
    nasa_sink s;
    ggeneric_mc filter1,filter2;
    // Create notes in pipeline graph.
    node1.create();
    node2.create();
    filter1.create();
    filter2.create();
    s.create(SELF);
    stats = s.initialize(iterations);
    // Set their delays.
    temp2=node1.set_delay(delay);
    temp2=node2.set_delay(delay);
    temp2=filter1.set_delay(fdelay);
    temp2=filter2.set_delay(fdelay);
    // Set their queue depths, these are varied.
    node1.set_atamm_queue(3);
    node2.set_atamm_queue(3);
    filter1.set_atamm_queue(3);
    filter2.set_atamm_queue(3);
    // Start timing.
    interval.start();
    for (i = 0; i < iterations; i++)
    {
        slave_arg j,k;
        j.arg1=interval.get_time();
        j = node1.one_arg(j);
        k=j;
        j = filter1.one_arg(j);
        j = filter2.one_arg(j);
        j = node2.one_arg(j);
        s.sink(j);
        j.delay=0;
    }
    long elapsed;
    elapsed = stats.TBO_msec;
    // Block and wait for statistics, then stop the timer.
    interval.stop();
    elapsed = interval.msec();
    elapsed = (elapsed) / iterations;
```

```
    delay /=1000;
    fdelay /=1000;
    int totalunits=((2*delay)+(2*fdelay));
    if (totalunits == 0) totalunits=1;
    printf(" mpipe %d %d work units %d Avg TIME = %d, TBO = %d, TBIO = %d\n",
        delay,fdelay,totalunits,elapsed,
        stats.TBO_msec/iterations,stats.TBIO_msec/iterations);
    // Clean up and destroy the objects.
    node1.destroy();
    node2.destroy();
    filter1.destroy();
    filter2.destroy();
    s.destroy();
}
```

NASA Graph-7

```
#define MAX_ITER 256
#include <stdio.h>
#include <libc.h>
#include <osfcn.h>
#include <signal.h>
#include <trace_object.h>
#include "ggeneric_mc.h"
#include "nasa_sink.h"

//#define output
//#define LOG

persistent mentat class n_graph {
        int iterations, current, sink_delay;
        int delay;
        mentat_timer interval;
        ggeneric_mc node1,node2,node3,node4,node5,node6,node7;
        nasa_sink graph_sink;
        trace_object log;
public:
        int main_loop(string* arg);
        slave_arg next(int next_val);
        int last(nasa_statistic result);
};

extern n_graph SELF;

slave_arg n_graph::next(int next_val) {
// First delay the right amount
mentat_timer temp;
unsigned val=temp.get_time();
int secs = sink_delay / 1000;
if (secs >0) sleep(secs);
slave_arg rval;
while ((temp.get_time()-val) < sink_delay);
// Then rtf to start the graph that is waiting on me
        // get_time() returns the "absolute" time in msec for this processor.
        int the_time=interval.get_time();
        rval.arg1=the_time;
        rtf(rval);
// Then build the next graph to execute.
        if (next_val<iterations)
        {
                slave_arg j,k,l,m,n,o,p;
                j=node1.one_arg(SELF.next(next_val+1));
                k=node2.one_arg(j);
                o=node6.one_arg(j);
                l=node3.one_arg(k);
                p=node7.one_arg(k);
                m=node4.two_arg(l,p);
                n=node5.two_arg(o,m);
                // "sink" the result
                graph_sink.sink(n);
        }
        else {
#ifdef output
                printf("last iteration\n");fflush(stdout);
#endif
        }
        return rval;
}

int n_graph::last(nasa_statistic result) {
        interval.stop();
```

```
#ifdef output
        printf("TBO, TBIO = %d,%d\n",
                result.TBO_msec/iterations,result.TBIO_msec/iterations);
#endif
        FILE *outfile = fopen("n_graph.output", "a+");
        fprintf(outfile,
                "delay/sink_delay/iterations/TBO/TBIO = %d %d %d %d %d\n",
                delay,sink_delay,iterations,
                result.TBO_msec/iterations,result.TBIO_msec/iterations);
        fclose(outfile);
        // ---------------------------
        // Calculate elapsed time
        long elapsed = interval.msec();
        // ---------------------------
        node1.destroy();
        node2.destroy();
        node3.destroy();
        node4.destroy();
        node5.destroy();
        node6.destroy();
        node7.destroy();
        graph_sink.destroy();
#ifdef LOG
        log.destroy();
#endif
        rtf(elapsed);
        SELF.destroy();
        return elapsed;
}
int n_graph::main_loop(string *arg) {

        sscanf((char*)arg,"%d %d %d",&iterations, &delay, &sink_delay);
#ifdef output
        printf("got init string %s\n", arg);
#endif
        current=1;
        int i,p_val,p;
        node1.create(SELF);
        node2.create(SELF);
        node3.create(SELF);
        node4.create(SELF);
        node5.create(SELF);
        node6.create(SELF);
        node7.create(SELF);
        int delay2=2*delay;
        int templ = 1;
        p=delay;
        // Set up delays as per figure 10
        templ=node1.set_delay(delay2);
        templ=node2.set_delay(delay);
        templ=node3.set_delay(delay);
        templ=node4.set_delay(delay2);
        templ=node5.set_delay(delay);
        templ=node6.set_delay(delay2);
        node6.set_atamm_queue(2);
        templ=node7.set_delay(delay);
        node7.set_atamm_queue(2);
        // ************************************************
        // This code may be a bit confusing.  It is really simple though.
        // Create the graph sink, then initialize it with the number
        // of iterations.  The initialize call will not return
        // until that number of "sink" calls have been made.
        // When they have, it returns a nasa_statistic to SELF.last.
        // SELF.last writes out the statistics, cleans up, and reports
```

```
        // our results.
        graph_sink.create(SELF);
#ifdef LOG
        log.create(SELF);
        string *logname = (string*) "logfile";
        log.open_log(logname);
#endif
        rtf(SELF.last(graph_sink.initialize(iterations)));
#ifdef LOG
        node1.set_log(log);
        node2.set_log(log);
        node3.set_log(log);
        node4.set_log(log);
        node5.set_log(log);
        node6.set_log(log);
        node7.set_log(log);
        graph_sink.set_log(log);
        SELF.set_log(log);
#endif
        // ************************************************
        // ----------------------------
        // Start up timer
        // ----------------------------
        // Code for graph construction here.
        interval.start();
        {
                slave_arg j,k,l,m,n,o;
                j=node1.one_arg(SELF.next(current));
                k=node2.one_arg(j);
                o=node6.one_arg(j);
                l=node3.one_arg(k);
                p=node7.one_arg(k);
                m=node4.two_arg(l,p);
                n=node5.two_arg(o,m);
                // "sink" the result
                graph_sink.sink(n);
        }
        return current;
}
```

```
#include<stdio.h>
#include"nasa_sink.h"

/*

class nasa_statistic {
public:
        long TBO_msec;
        long TBIO_msec;
};

persistent mentat class nasa_sink {
        int count;
        int first;
        nasa_statistic stats;
        mentat_timer TBO;
  public:
        void sink(int argument);
        nasa_statistic initialize(int num_to_expect);
};


*/

void nasa_sink::sink(slave_arg argument) {
        // Compute TBO
        if (first==0) {
                TBO.stop();
                stats.TBO_msec+=TBO.msec();
        }
        else first=0;
        // Go ahead and reply.
        rtf(0);
        // Now compute the TBIO
        // get_time() returns the "absolute" time in msec for this processor.
        long temp= (TBO.get_time()-argument.arg1);
        stats.TBIO_msec+=temp;
        // Restart TBO
        TBO.start();
        // Decrement counter, on zero do two returns, cleaning up
        // the stack and replying to the initialize
        count--;
        if (count==0) {
                rtf(stats);
        }
}

nasa_statistic nasa_sink::initialize(int num_to_expect) {
        // Set up fields, but don't rtf, that will be done later.
        count = num_to_expect;
        stats.TBO_msec=0;
        stats.TBIO_msec=0;
        first=1;
        return stats;
}
```

Motorola 68020 (Sun 3/60)

gcc -O -fstrength-reduce -fomit-frame-pointer (Version 1.37)

Benchmark	Run time	Compile time	Comments
eqntott	288.0s	71.5s	
espresso	***.*s	403.6s	core dumps
gcc	***.*s	1293.0s	core dumps
li	1843.1s	122.5s	

cc -O3

Benchmark	Run time	Compile time	Comments
eqntott	356.5s	168.6s	
espresso	878.6s	1040.3s	
gcc	418.5s	2416.5s	-O2, will not compile with -O3
li	2030.1s	293.9s	

vpcc/vpo -LOGCMSV (compiled with cc -O3 (need to try gcc and vpcc/vpo))

Benchmark	Run time	Compile time	Comments
eqntott	322.7s	176.0s	
espresso	825.0s	1110.4s	
gcc	414.9s	3795.2s	
li	1816.8s	255.8s	

VAX-11 8600

gcc -O -fstrength-reduce (Version 1.37)

Benchmark	Run time	Compile time	Comments
eqntott	254.4s	39.5s	
espresso	***.*s	***.*s	cannot compile (stdlib.h problem)
gcc	368.5s	548.1s	
li	1714.6s	56.4s	

cc -O

Benchmark	Run time	Compile time	Comments
eqntott	417.4s	29.3s	
espresso	761.2s	208.2s	
gcc	400.1s	703.9s	
li	1704.3s	60.7s	

vpcc/vpo -LOGCMS (compiled with gcc (need to try vpcc/vpo))

Benchmark	Run time	Compile time	Comments
eqntott	277.7s	122.6s	
espresso	599.3s	860.6s	
gcc	395.6s	3212.1s	
li	1526.2s	198.1s	

VAX 11 750

vpcc/vpo -LOGCMSV (compiled with gcc (need to try vpcc/vpo))

Benchmark	Run time	Compile time	Comments
eqntott	1770.4s		
espresso	3880.3s		
gcc	2548.7s		
li	10573.8s		

Intel 30386/30387

MetaWare High C Version 2.2c -O -Hnoansilib -U__STDC__

Benchmark	Run time	Compile time	Comments
eqntott	251.7s	60.2s	
espresso	473.6s	418.7s	
gcc	***.*s	1700.7s	core dumps
li	1023.4s	101.9s	

cc -O

Benchmark	Run time	Compile time	Comments
eqntott	341.5s	39.6s	
espresso	584.3s	241.5s	
gcc	278.2s	914.3s	
li	1203.2s	83.2s	

vpcc/vpo -LOGCMSV (compiled with vpcc/vpo (best times))

Benchmark	Run time	Compile time	Comments
eqntott	326.1s	100.4s	
espresso	514.0s	845.7s	
gcc	254.3s	3074.6s	
li	1076.0s	178.7s	

SPARC (SPARC Station 2)

gcc -O -fstrength-reduce -fomit-frame-pointer -fdelayed-branch (Version 1.37)

Benchmark	Run time	Compile time	Comments
eqntott	64.9s	15.5s	
espresso	136.9s	84.6s	
gcc	***.*s	295.5s	core dumps
li	399.0s	26.1s	

cc -O3

Benchmark	Run time	Compile time	Comments
eqntott	57.1s	39.8s	
espresso	124.1s	220.0s	
gcc	76.5s	550.1s	-O2, will not compile with -O3
li	339.7s	62.2s	

vpcc/vpo -LOGCMSVF (compiled with cc -O3 (best times))

Benchmark	Run time	Compile time	Comments
eqntott	48.6s	46.1s	
espresso	124.4s	261.5s	
gcc	79.5s	891.6s	
li	343.6s	64.4s	

MIPS R3000 (SGI)

gcc -O -fstrength-reduce -fomit-frame-pointer -fdelayed-branch -U__STDC__ 1.39

Benchmark	Run time	Compile time	Comments
eqntott	59.7s	23.6s	
espresso	106.1s	122.2s	
gcc	***.*s	***.*s	cannot compile (varargs problem)
li	285.6s	42.9s	

cc -O2 -G 0

Benchmark	Run time	Compile time	Comments
eqntott	55.1s	24.4s	

espresso	100.8s	123.0s
gcc	72.1s	528.3s
li	258.6s	36.3s

vpcc/vpo -LOGVCMSV (compiled with cc -O2)

Benchmark	Run time	Compile time	Comments
eqntott	50.9s	35.4s	
espresso	108.1s	235.6s	
gcc	76.5s	799.2s	
li	275.1s	55.8s	

Motorola 88100

Green Hills Software C-88000 1.8.4m17 -O (OPTIM=LM)

Benchmark	Run time	Compile time	Comments
eqntott	131.0s	44.6s	
espresso	191.4s	260.3s	
gcc	110.5s	633.9s	OPTIM=M (Won't compile with LM)
li	446.5s	68.6s	

vpcc/vpo -LOGCMSVFI (compiled with cc -O)

Benchmark	Run time	Compile time	Comments
eqntott	97.1s	58.1s	
espresso	189.8s	367.9s	
gcc	113.1s	1404.8s	
li	454.2s	91.5s	

```
#ifndef GGENERIC_MC_H
#define GGENERIC_MC_H

struct slave_arg {
        int arg1;
        int delay;
};

persistent mentat class ggeneric_slave {
  public:
    slave_arg   one_arg(slave_arg arg1);
    slave_arg   two_arg(slave_arg arg1, slave_arg arg2);
    slave_arg   three_arg(slave_arg arg1, slave_arg arg2, slave_arg arg3);
};

persistent mentat class ggeneric_mc {
    int delay;
        int count;
        int outstanding;
        ggeneric_slave slave;
  public:
    int set_delay(int delaa);
    slave_arg   one_arg(slave_arg arg1);
    slave_arg   two_arg(slave_arg arg1, slave_arg arg2);
    slave_arg   three_arg(slave_arg arg1, slave_arg arg2, slave_arg arg3);
    int print(int dummy);
    void set_atamm_queue(int);
    slave_arg   keep_count(slave_arg arg1);
};
#endif
```

```
#include "ggeneric_mc.h"
class nasa_statistic {
public:
        long TBO_msec;
        long TBIO_msec;
};

persistent mentat class nasa_sink {
        int count;
        int first;
        nasa_statistic stats;
        mentat_timer TBO;
  public:
        void sink(slave_arg argument);
        nasa_statistic initialize(int num_to_expect);
};
```

```
#include "ggeneric_mc.h"
#include<stdio.h>

#ifdef ATAMM
extern int atamm_threshold;
#else
int atamm_threshold = 10000;
#endif

int ggeneric_mc::set_delay(int delaa) {
        delay = delaa;
        atamm_threshold = 1;
        count=0;
        outstanding=0;
        rtf(delay);

}
void ggeneric_mc::set_atamm_queue(int size) {
        atamm_threshold=size;
        rtf(0);

}


slave_arg ggeneric_mc::keep_count(slave_arg arg1) {
        outstanding--;
        //printf("entering generic keepcount \n");fflush(stdout);
        rtf(arg1);
        return arg1;
}

slave_arg ggeneric_mc::one_arg(slave_arg arg1) {
        // increment outstanding
        outstanding++;
        arg1.delay=delay;
        //printf("entering generic one arg\n");fflush(stdout);
        // check against queue depth
        if (outstanding > atamm_threshold) {
                //printf("too many outstanding %d %d\n",
                        //outstanding,atamm_threshold);fflush(stdout);
                // Do something here to wait until another has
                // been registered.
                // Want a select accept on keep count
{
        // This code fragement is a call into the Mentat RTS, and
        // is implements a select/accept, a feature not currently
        // supported by the compiler.
    int predicate_number;
    mentat_message *msg1, *msg2, *msg3;
    msg1 = msg2 = msg3 = NULL;
    predicate_manager *predicate_manager_instance = new predicate_manager(1);
    predicate_number = predicate_manager_instance->enable_operation(0, 107, 1);
        predicate_number = -1;
                //printf("before block\n");fflush(stdout);
        predicate_number = predicate_manager_instance->block_predicate(&msg1, &msg2, &msg3
);
                //printf("past block\n");fflush(stdout);
            {

                slave_arg arg1 = RESOLVE_MSG(slave_arg, msg1);
                slave_arg result = keep_count(arg1);
                delete msg1;
            }
    delete predicate_manager_instance;
```

```
        // End of select/accept code fragement.
        }
        }
        // Now do tail recursive call to SELF.keep_count()
        rtf(SELF.keep_count(slave.one_arg(arg1)));
        count++;
        return arg1;
}


slave_arg ggeneric_mc::two_arg(slave_arg arg1, slave_arg arg2) {
        // increment outstanding
        outstanding++;
        arg1.delay=delay;
        // check against queue depth
        if (outstanding > atamm_threshold) {
                //printf("too many outstanding\n");fflush(stdout);
                // Do something here to wait until another has
                // been registered.
        }
        // Now do tail recursive call to SELF.keep_count()
        rtf(SELF.keep_count(slave.two_arg(arg1,arg2)));
        count++;
        return arg1;

}


slave_arg ggeneric_mc::three_arg(slave_arg arg1, slave_arg arg2, slave_arg arg3) {
        // increment outstanding
        outstanding++;
        arg1.delay=delay;
        // check against queue depth
        if (outstanding > atamm_threshold+3) {
                //printf("too many outstanding\n");fflush(stdout);
                // Do something here to wait until another has
                // been registered.
        }
        // Now do tail recursive call to SELF.keep_count()
        rtf(SELF.keep_count(slave.three_arg(arg1,arg2,arg3)));
        count++;
        return arg1;
}


int ggeneric_mc::print(int dummy) {

        printf("count is %d\n",count);
        rtf(dummy);

}
```

```
#include "ggeneric_mc.h"
#include<stdio.h>
extern int atamm_regular;



slave_arg ggeneric_slave::one_arg(slave_arg arg1) {
        // First delay the right amount
#ifdef ATAMM
        atamm_regular=1;
#endif
        int count_len = arg1.delay * 550;
        // Simulate computation.
        for (int i=0;i<count_len;i++);
        rtf(arg1);
        instance_manager im;
        im.mark_available(SELF);
        return arg1;
}


slave_arg ggeneric_slave::two_arg(slave_arg arg1, slave_arg arg2) {

        // First delay the right amount
#ifdef ATAMM
        atamm_regular=1;
#endif
        int count_len = arg1.delay * 500;
        for (int i=0;i<count_len;i++);
        rtf(arg1);
        instance_manager im;
        im.mark_available(SELF);
        return arg1;
}


slave_arg ggeneric_slave::three_arg(slave_arg arg1, slave_arg arg2, slave_arg arg3) {

        // First delay the right amount
#ifdef ATAMM
        atamm_regular=1;
#endif
        int count_len = arg1.delay * 500;
        for (int i=0;i<count_len;i++);
        rtf(arg1);
        instance_manager im;
        im.mark_available(SELF);
        return arg1;
}
```