

Mode control with GEDAE: a real time distributed signal processor control application.

Douglas Swanson

BAE SYSTEMS Avionics

Sensor Systems Division,

Crewe Toll House, Ferry Road,

Edinburgh EH5 2XS, U.K.

douglas.swanson@baesystems.com

ABSTRACT

This paper shall discuss a complex GEDAE application that has been developed by BAE SYSTEMS for use on the CAPTOR Tranche 2 radar programme. The application is a processor for a multimode radar with a distributed target hardware platform. Specifically, the paper concerns the signal processor controller, and a mode control framework, both of which have been developed under GEDAE, as a single application. The paper shall present a description of the implementation of this application, and is an important step in proving the viability of GEDAE for developing control-based applications such as this. It does this by providing a demonstration of a complex control implementation in a large-scale project.

Keywords: GEDAE, CAPTOR, control, multimode, framework, distributed.

1. INTRODUCTION

CAPTOR Tranche 2 Radar is a multimode radar developed for the Eurofighter Typhoon aircraft. The signal processor is designed to operate on a distributed, multi-target, hardware platform. The processing engines are required to process radar data sets in a round robin sequence to maximise performance, and increase throughput.

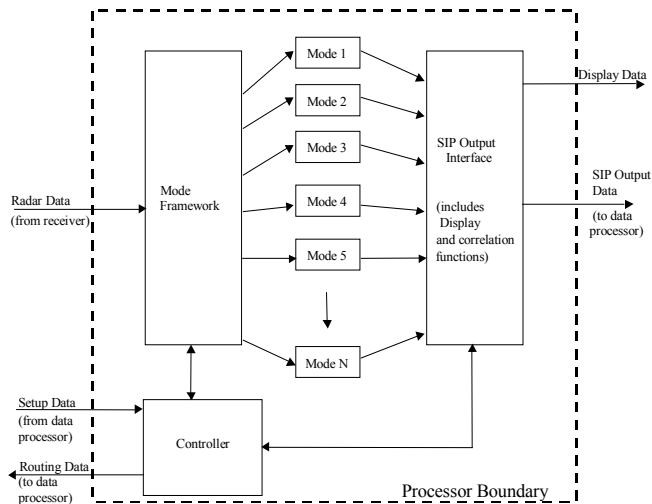


Figure 1. Example hardware configuration

The controller software resides on a single processor, and is required to manage mode processing on a distributed set of processing engines. The output from these disparate engines is subsequently collated on a single correlation processing engine, from where it is output from the signal processor application to an external entity. It is vital to pay proper regard to this distribution of the signal processing elements when designing a control application, since the management and routing of data and control parameters across the hardware is a fundamental task of the controller and framework application.

The controller provides an interface to the radar data processor entity, which provides a stream of mode control parameter sets to the signal processor, in addition to a number of asynchronous commands that must be handled. The mode control parameter sets define the radar mode for which the receiver is collecting data, and which must therefore be processed by the signal processor.

2. DISTRIBUTION

In order that the radar modes can be correctly distributed, it is necessary to know how many signal processing engines are available, how many modes must be modelled, and the processor requirements of each mode. These parameters are represented in the following figure as Tprocs (which is itself defined by a number of processors on a number of cards), Nmodes, and Nppi, respectively. These are supplied to the model by the user, and used to calculate the maximum possible number of instances of

each mode that can be processed simultaneously. From these, it is possible to calculate the maximum number of possible instances that the system can support, N_{inst} , for each mode m .

```

const int Tprocs = 6
const int Mode_Ids[] = {1,2,3,4}
const int Nmodes = 4

range p = 0..Tprocs-1
range m = 0..Nmodes-1

const int Nppi[] = {1,2,4,1}
const int Ninst[m] = ((Nppi[m]>0)?Tprocs/Nppi[m]:0)

const int InstMap[m][p] = (m)LeadVPESLim[p]
m
const int LeadVPES[p] = ((Nppi[m]>0)?p*Nppi[m]:-1)
m
const int LeadVPESLim[p] = ((([m]LeadVPES[p]<Tprocs)&&([m]LeadVPES[p]>=0))?[m]LeadVPES[p]:-1)

```

Figure 2. Configuration formulae

This calculation applied across all of the available modes will give the total number of mode instances; to any of which the controller may be required to send control parameters. In the example outlined in Figure 2, there are four modes and six processors. All of the modes require different numbers of processors to run one instance, except mode indices 0 and 3, both of which require one processor. The impact of this can be seen in Figure 3, which shows that mode indices 0 and 3 can have six possible instances, while the other modes have less instances, since they require extra processors.

```

Value of Ninst =
0: 6
1: 3
2: 1
3: 6

```

Figure 3. Ninst

It is therefore possible in this example to have 16 possible instances in total. By selecting a single one of these instances the controller is making a decision about which mode graph the parameter set will be input to, and also which instance of the mode is to process the corresponding data set. The decision of which mode graph should be run is determined by the mode number of the control parameter set received by the controller, but, the decision of which instance of

the mode should be selected is performed by a processor load-balancing algorithm in the controller. Each instance of a mode is mapped onto the different processing engines, and so multiple instances of the mode will run in parallel within the signal processor.

3. LOAD BALANCING

The load-balancing algorithm is used to calculate which set of processing engines will be available soonest, in order to process a radar mode as quickly as possible. To achieve this, the controller maintains a vector containing the time remaining until each individual processor is expected to have completed its current processing task. To continue using the example hardware configuration from Figure 2, this vector will have six elements, since there are six signal processors. The algorithm is also supplied with another vector $InstMap$, which defines the starting processor for each instance of each mode, and the number of processors required for each mode. $InstMap$ is automatically calculated from the number of processors, the number of modes,

```

Value of InstMap =
0 0: 0
0 1: 1
0 2: 2
0 3: 3
0 4: 4
0 5: 5
1 0: 0
1 1: 2
1 2: 4
1 3: -1
1 4: -1
1 5: -1
2 0: 0
2 1: -1
2 2: -1
2 3: -1
2 4: -1
2 5: -1
3 0: 0
3 1: 1
3 2: 2
3 3: 3
3 4: 4
3 5: 5

```

Figure 4. InstMap

and the number of processors required to process one instance of the mode. The calculation of $InstMap$ is shown in Figure 4. The three columns represent the mode number, the instance number, and the processor on which that instance may be run. Notice that the values of $InstMap$ for mode index 1 are 0, 2, 4, -1, -1, and -1. This is because mode index 1 requires two processors to execute, and so instances of the mode may only be started on processors 0, 2, and 4, not 1, 3, or 5. The remaining three instances of mode 1 are set to -1, which signifies an invalid processor number, since there are only three valid instances of this mode. Since it is not possible for a mode to require less than one processor, there can only be a maximum of p instances of a mode, where p is the total number of processors – in this case six.

From this and the vector of times until completion, it is possible to calculate which processor will become ready last in each instance. This is considered to be the earliest time the set of required processors will be available. The instance whose processors will be ready soonest is then selected, and the estimated processing time

of the mode added to the times until completion for the processors involved in the mode. This updated vector of times until completion is then fed back around as an input to the load-balancing algorithm ready for the next data set to be calculated. In this way, the controller ensures that the available data sets will be processed as quickly as possible, keeping latency to a minimum. The lead processor in the set of processors is designated as the one that the control parameters must be sent to, and therefore the one that receives the radar data associated with this set of control parameters. The controller informs the external data processor of what decision has been made, and the data processor makes this information available to the receiver to allow it to route the radar data appropriately, as can be seen in Figure 1.

4. RADAR DATA INTERFACE

The controller must route the parameter set to the correct instance of all possible instances of all possible modes. In other

words, the controller must determine which of the 16 possible mode instances in this example is the correct one. A further set of equations is defined to facilitate this data routing process.

The controller uses the InstMap to establish which processor the selected instance exists on, and sends the parameter set to the appropriate processor. At this point in the processing, the functionality is distributed to the processing engines, and is no longer executing on the single controller processor. This is conceptually the end of the control application, and entering the domain of the mode framework. The framework incorporates an interface to the receiver, through which the radar data will arrive. In this way radar data is only sent to the relevant processor, helping to minimise bus traffic, and avoiding unnecessary impact on bandwidth and interface memory space.

```

const int Nmodes
const int Tprocs
const int Nppi[]
const int Ninst[]
const int Inst2Proc[][]

range m = 0,.,Nmodes-1
range m1 = 0,.,Nmodes-1
range p1 = 0,.,Tprocs-1
const int Ainst[m] = sum<Ninst,m>

m,p
const int work1[p1] = <Inst2Proc[m][p1]==p>?1:0
p
const int ModeOnInst[m] = sum<[m][p]work1,Tprocs>
const int NinstsOnProc[p] = sum<[p]ModeOnInst,Nmodes>
p
range mpp = 0,.,NinstsOnProc[p]-1
p1,m
const int work2[p] = <Inst2Proc[m][p]==p1>?p:0
m
const int work2a[p] = sum<[p][m]work2,Tprocs>
p,m1
const int PI2M1[m] = <<[p]AMonI[m]==m1>&&<[p]ModeOnInst[m]==1>?m:0
const int PI2M[p][m] = sum<[p][m]PI2M1,Nmodes>
p
const int P2MI[m][p] = Ainst[PI2M[p][m]]+[PI2M[p][m]]work2a[p]
const int AMonI[m] = sum<[p]ModeOnInst,m>

```

Figure 5. Instance routing equations

5. INSTANCE ROUTING

Figure 5 shows the formulae used to calculate this mapping of instances of modes with processors, and an explanation of how this happens follows. This is critical to the successful operation of the application, since this eliminates any superfluous transfers of the data sets, which tend to be of large sizes. This is imperative in developing any real time application.

PI2M is calculated to provide an association of each instance on each processor to the mode number that instance relates to. Since instances of different modes may have different processor

requirements, it is necessary to do this for each mode on the processor. Work1 represents which of the processors each instance of each mode may be run on, and is used to calculate the ModeOnInst values. ModeOnInst is a family of TProcs vectors, each with Nmodes elements showing whether the mode may be run on the processor. It contains the same information as work1, but illustrated in a clearer form. ModeOnInst contains a logical 1 or 0 flag, and this is used to calculate AMonI, which sums these flags, and returns the instance numbers of each mode on each processor. For example, processor 1 will have two modes available to run on it (modes 1 and 2 cannot have an instance on processor 1); this will return a value of ModeOnInst for processor 1 of {1,0,0,1}, and a value of AMonI of {0,1,1,1}. It can readily be seen that ModeOnInst also allows the framework to establish how many instances can be run on each processor, which is stored as NinstsOnProc. This is in turn used to create a range, mpp, for each processor, which covers each possible instance for each processor. AMonI and ModeOnInst are logically ANDed together so that the framework can establish what mode an instance must be, if it may be run on any particular processor. This intermediate stage PI2M1 is then summarised as a matrix of processors and modes PI2M, which is in turn converted into an overall matrix of processors*modes, P2MI, placing each instance into the context of all the possible modes in the system. As can be seen in Figure 6, that this assigns one of the 16 possible instance numbers to each mode instance running on each processor.

PI2M is then used to route the radar data to a particular mode instance on a processor. Since the framework now knows where the data should be processed, it is possible to read in the radar data in the right place. Waiting until now to acquire the radar data means that a large data set does not have to be transferred internally any more often than is necessary, helping to keep the performance good enough for real time signal processing.

6. MODE GROUPING

At this point, it is useful to consider the operation of the radar modes in slightly more detail than has hitherto been the case. The framework quite rightly does not concern itself with the functionality of the modes, beyond what is required to interface with them correctly. In order that the outputs from the modes can be correctly modelled, however, it is useful to consider the modes belonging to one of two basic types - operational or calibration. The output from the mode must be handled in one of two ways: if the mode is an operational mode, the output must be ordered and sent for correlation; or else if the mode is a calibration mode, the output must be used to provide calibration data for following modes. It is useful to separate these two types of mode as quickly as possible and keep them apart, and the framework does this next.

Each of the total available instances is associated with either an operational or a calibration mode, and the parameter set and radar data are routed appropriately. In our example, mode indices 0, 1, and 2 are defined as operational, and mode 3 is a calibration mode. The instance output from each processor that relates to the calibration modes are routed to the calibration outputs, while the operational outputs are mapped to operational_params outputs. This can be seen in Figure 6. This also helps to demonstrate the concept that an instance on each processor does not relate to a mode index directly. Processor 0 can run any of the modes and so

it has four outputs (the first four on the left hand side), each mapping directly to a radar mode. Processor 1, however, cannot run an instance of mode index 1 or 2, as discussed previously and so there only two instances, although these are still numbered 0 and 1. This means that mode index 0 still maps to radar mode 0, but instance 1 actually maps to the next mode which can be run on the processor. In this case, this is mode 3, which is a calibration mode. This is clearly demonstrated by instance 1 on

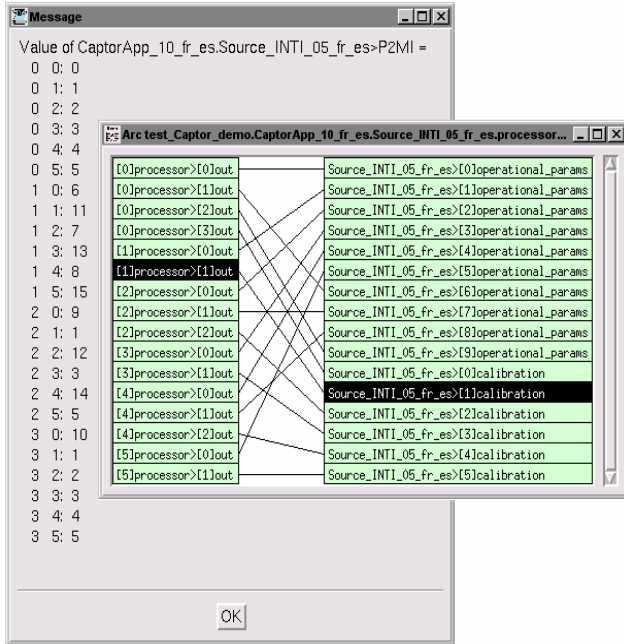


Figure 6. P2MI and operational/calibration mode routing

processor 1 being routed to a calibration output, and not an operational output, as would be expected for mode index 1.

The operational modes are further grouped into mode types, and ultimately into multiple instances of individual radar modes. This grouping allows persistent state to be maintained at several levels, depending on whether it must be shared between instances of a single mode, or between instances of similar modes. The interfaces to the modes are as consistent as possible to ease the progressive integration of radar modes. The modes themselves are developed in parallel with the framework and controller application, and can be added into the framework that already exists for them whenever they become ready. The controller and framework model recalculates all of the possible instances and routes based upon the correct setting of the number of modes, how many processors each mode requires, and which group they are a member of.

7. CALIBRATION MODE HANDLING

Calibration mode outputs are not passed out of the signal processor; instead, their outputs are used to refine the processing of subsequent data sets. This means that the framework must route the outputs from these modes back into the controller. The controller can then add this data to the control parameters for the next data set to be processed.

This is done with a custom CalMerge box, which has three non-deterministic inputs, which can be seen in Figure 7. One of these is a stream containing the fed back calibration data, from a previously completed calibration mode; the second is an externally generated command to perform a reset of the calibration data; and the third is the data associated with such a reset. The second and third streams work as a pair in terms of the time of data arrival. The reset command comes periodically from the data processor, and is managed by the controller to become this input to the calibration merge. None of these streams are necessary for the box to output a calibration table to be packaged up with a control parameter set, and used to control a mode processing instance. If any data does arrive at the CalMerge box it is used to update the locally held calibration state, with any external updates taking priority over a fed back calibration data set. When the controller has received a set of control parameters from the data processor, it is ready to include the latest calibration

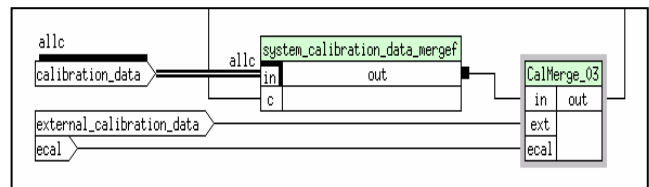


Figure 7. Calibration state merge

data. The locally held calibration data is combined with the control parameter set and routed out to the signal processing engine selected by the controller during load balancing.

8. OPERATIONAL MODE HANDLING

The operational mode outputs must be passed on for correlation in the order in which they were processed, but there is no guarantee that the order of completion will be the same as the order in which the processing was started. This is because of variations in the processing time, caused by variations in the data set sizes, and the enabling and disabling of various processing options by the control parameter set. It is therefore necessary to include an ordering box, to ensure the correlation processor is given the data sets in the correct sequence. This box takes as its input a family of parameter sets, and a family of mode output data. The families both have the total number of operational mode instances as the size of the family. In this example, there are 10 such instances. The 10 instances can be observed in Figure 6. The instance number that the controller generated to specify the particular mode instance on a particular processor is passed into this ordering box as a control parameter. Those instances relating to calibration modes are filtered out of this stream, since no calibration mode outputs will be passed to the ordering box, and the remaining instances are used to control the merging of the family inputs. In this way, the ordering box will consume the family inputs in the same order as the mode instances were initiated, and the resultant ordered stream of output data sets can be passed onto a correlation processor. The controller and framework task is now complete for a particular operational mode instance.

9. PERSISTENT STATE

Certain operational modes have a requirement to maintain a filter state from one instance of the mode to the next. If these state values are to be calculated by data set (n), those values

calculated by the previous data set (n-1) are required, and the result will be used by the following data set (n+1). This can be modelled in GEDAE using a delay on a feedback loop. This will ensure that the state input is delayed by a definable number of data sets, and is therefore available on subsequent firings of the mode. This mechanism is not satisfactory for a distributed environment, since there is no information telling the controller which of the possible processing instances of the mode the state should be acquired from, or which processing instance will require the next output state data. In other words, the instance n-1, and instance n+1 may exist on different processors, and can therefore appear on, or need to be routed onto, any of the family of processors to which the feedback connects. This could be achieved by having the framework route the filter state outputs from each processor back to the controller. This solves the immediate problem, in that there is no longer any decision about where the data comes from, or goes to. This could then be combined with the control parameter set and sent to the next instance as it was required, in a manner very similar to the calibration data feedback. This solution is not ideal for filter state feedback, since there is no such external update to be modelled, and so there is no functional reason for sending it to the

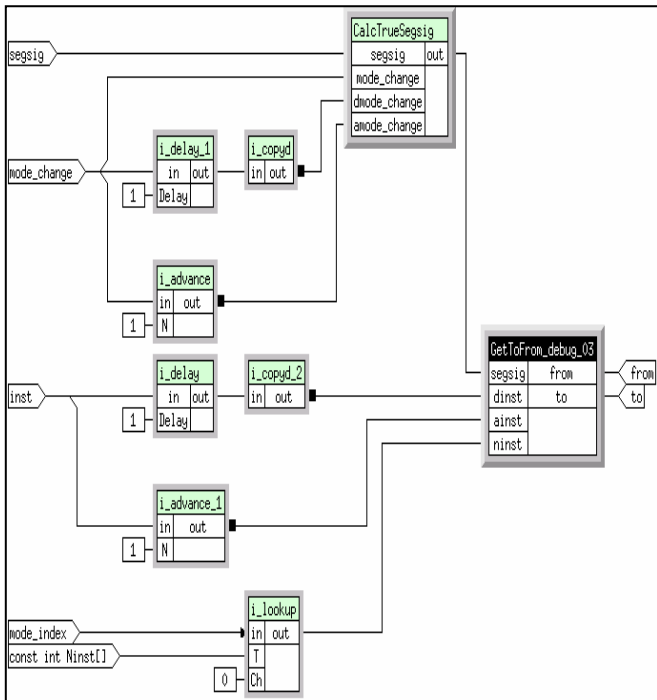


Figure 8. State routing parameter calculation

controller. It also serves to increase the data bus traffic, and degrade the overall performance of the signal processor.

In order to minimise latency, it is better to use a scheme such as has been incorporated in this model, shown in Figure 8. This uses “to” and “from” parameters to identify the previous and next instance number. These numbers are calculated by the controller, based on the instance selected for each set of control parameters.

Since the “from” parameter is the number of the previous instance (data set n-1), it is possible to calculate this at the time of data set n. The instance to which to send the state calculated by n

cannot be calculated until the controller has handled the next set of control parameters. This means the “to” parameter cannot be sent along with the rest of control parameters for n, and must arrive asynchronously to the instance. This solution migrates the need to delay the state data directly, to a delay on the “to” parameter in the controller. This has the advantage of not being mapped onto distributed processing engine hardware, only onto the single controller engine. Sending the “to” parameter separately means there is no adverse impact on performance, since it is not needed until after the end of the mode processing, when the output is to be routed onto the next instance. By explicitly sending the from and to instance numbers it is possible to send the state data directly to those processors which require it, keeping data transfers to a minimum.

The flattened graph extract shown in Figure 9 contains four instances of a single mode, which has a persistent state feedback loop. The bottom right box in each mode instance branches an output back to the start of the next instance, and so it connects to each instance, including itself. This is because it is quite possible for the next data set that requires the state information to appear on the same processor set as the previous one. The third box from the left on each mode instance is the merged input for the state filter. This box accepts input from any of the possible mode instances that feed into it. The “from” parameter is supplied to tell the merge which of the four possible instances has run previously, and so where to acquire the state data from. In a similar fashion, the “to” parameter is supplied to the branch box,

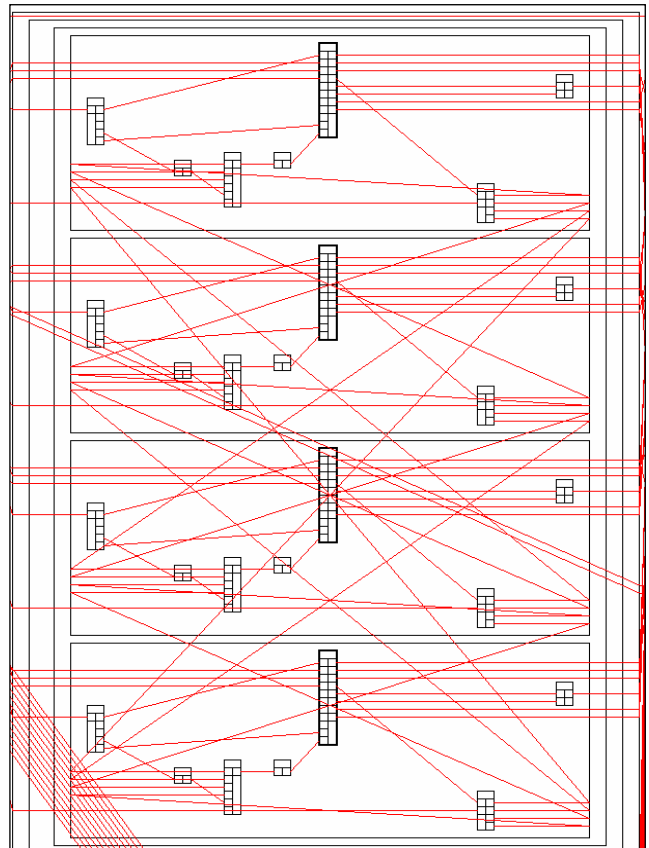


Figure 9. State feedback

albeit not until the next data set is running. This tells the

framework where to route the newly calculated state data, in order that the next instance can be processed correctly.

10. FUTURE DEVELOPMENT

The controller and framework model has been built, but has not yet been modified to take full advantage of a number of important new GEDAE concepts. The most interesting of these future developments will make proper use of the related concepts of segmentation and exclusion. This will remove the need for a single processing engine to allocate memory for all of the radar modes simultaneously. Since the controller will prevent multiple radar modes being started on the same processing engine, it is not necessary for a processor to reserve memory for all of the possible radar modes at one time. The concept of exclusive segments will allow the scheduler to allocate only enough memory to contain the largest of the modes within a defined set of graphs or boxes, called a segment. This will have the effect of reducing the overall memory footprint of the signal processor, and improve the portability between hardware platforms, by lessening the basic hardware requirements of the system.

The third important concept to incorporate in the future is that of state. State boxes will allow the sharing of data between boxes and graphs within a segment, without the need to explicitly flow the data from one box to the next. This will significantly simplify the handling of persistent data. The flowgraphs and flattened graphs will become less convoluted, and consequently easier to maintain and debug. The successful implementation of this is reliant on the careful selection of the segmentation levels within the graph, and so it can be seen that these three concepts are very closely interrelated.

11. CONCLUSION

It can be seen from the model that it is perfectly possible to construct complex control applications using GEDAE. Such an

approach allows control and data processing to be developed in a single integrated environment. This simplifies the testing of the data driven processing, and the integration of, in this case, further radar modes. By maintaining a consistent interface between the mode framework, and the independently developed radar mode algorithms, the task of integrating new modes is made significantly simpler, and faster, than in a more conventional embedded implementation. GEDAE can also be used to simulate external entities and events, providing the control application with appropriate stimuli, and assisting in the process of developing a test suite.

It can be seen that it is possible to accurately model multimode operation, and control the distribution of these modes across a scalable set of processors. The framework and controller application discussed here has been demonstrated as being capable of managing persistent state, which can be applied at different levels of graph grouping, from firing to firing. This concept can have applications in a number of different scenarios, across a range of applications. It has also been shown that non-deterministic feedback, and asynchronous update can be modelled, as is the case with the system calibration data. In summary, developing a GEDAE based control application is an achievable and sensible activity, which can be combined with GEDAE-produced data driven applications, to provide a single development environment for complex systems.

12. ACKNOWLEDGMENTS

The author wishes to express thanks for the support of all those colleagues at SSD Crewe Toll, EADS Ulm, and Blue Horizon Development Software, who contributed in any way to this work..