Magyar Kutatók 7. Nemzetközi Szimpóziuma
7th International Symposium of Hungarian Researchers on Computational Intelligence

Generic Software Architecture for Startup Sequencing and Monitoring Diagnostics/Prognostics

William Franklin

University of Maryland University College, USA
franklin2623@sbcglobal.net

Abstract: A software architecture is proposed for Diagnostics and Prognostics that makes all the service functions service functions generic and derivable from a common architecture. This allows all of the processing to be more automated, standardized and validated. The proposed architecture consists of design patterns and state machines for the control logic of smart sensors and intelligent agents that facilitate the diagnostics/prognostics process. Object Oriented features such as polymorphism and inheritance are utilized to establish heuristic abstractions that are generic across all service modes of operation. These abstractions allow for the inheritance of state chart behavior from base classes to derived classes and the customization of these state charts as deemed appropriate for each derived class. Although this paper utilizes this architecture to diagnostics and prognostics it is applicable to other types of services that utilize control logic with state machines. These features are presented using class and state chart diagrams in the Rational Rose Unified Modeling Language (UML).

1 Start Up

The StartUpMasterSequence acts as the control creator of all the processing and is intended to manage the sequencing and concurrency in the spawning of processes. Process spawning could be serialized as necessary to preempt memory and resource thrashing (page faults displayed frequently) that might occur if the processes in contention were spawned concurrently. Memory and resource thrashing are minimized by the proper choice of the OS. Figure 1 illustrates the Domain Diagram for this.

The StartUpMasterBuilder utilizes a **Builder** Design Pattern [1-3] that separates the construction of a complex object from its representation. This particular pattern has the following heuristics:

– Construction process allows different representations for the object constructed.

  'Build up by parts rather than whole.'

- Algorithm for creating object should be independent of the parts that make up the object and how they are assembled. **Example – building a house (generic).**
- Creator Pattern but with **low class coupling** whereas Creator Pattern has **high class coupling**.
- It constitutes an elaborate initializer.

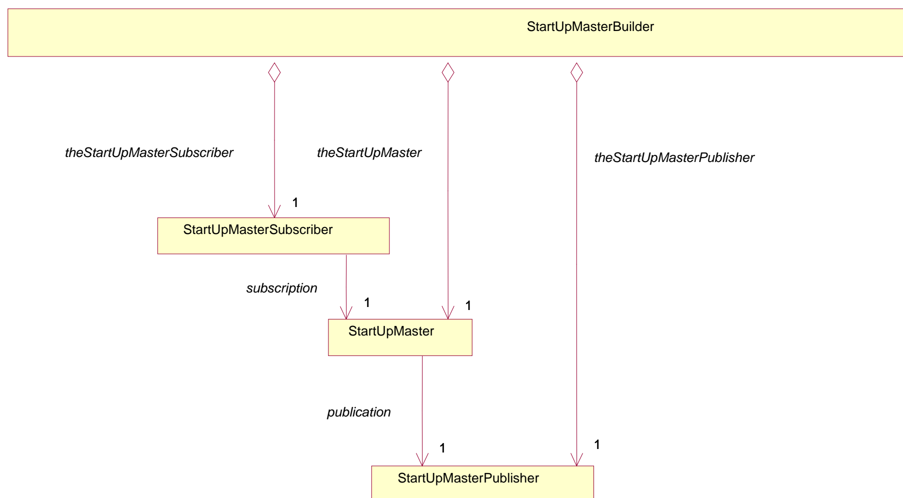

Figure 1
StartUpMasterBuilder Domain Diagram

We will see more applications of the **Builder** Design Pattern in ensuing diagrams and applications.

The StartUpMaster component controls the order of spawning and initializing of all software components in the embedded control logic. This start up sequencing addresses technological requirements imposed by the OS, ROSE RT compiler, device drivers, as well as functional dependencies that the involved components have on data that must be acquired at start up prior to normal operation. Figure 2 StartUp Master Composition in StartUpMasterPackage illustrates the ensuing components that the StartUpMaster kicks off. This shows the class relationships but the real mechanics will be detailed in the ensuing state charts of each class.

Because the spawning of processes is intensive in the manner it demands memory and other resource allocations from the OS, the StartUpMaster manages the sequencing and concurrency of the process spawning. The StartUpMaster determines the concurrency of the spawning of individual processes as is required for timeliness and efficiency in the startup of processes.

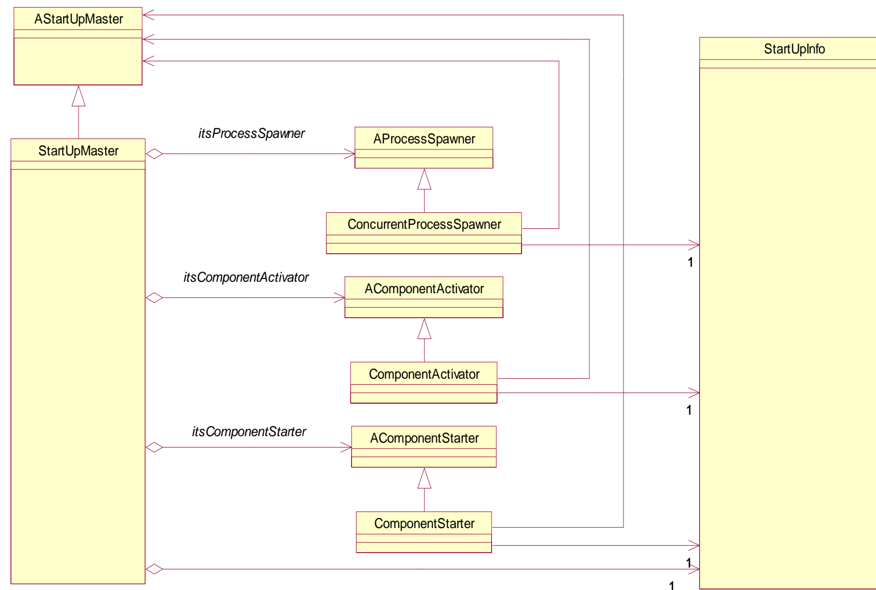

Figure 2

Start Up Master Composition in StartUpMasterPackage

The solution for both of these problems is to strictly control the sequencing for the creation and declaration of subscriptions and publications across all software components. The following sequence eliminates these two problems:

1) Establish bootstrap subscriptions and publications in the components so that they can communicate with the StartUpMaster and so the StartUpMaster can control the sequencing of the creation of the other subscriptions and publications used by the components.
2) Disable message declarations for each component.
3) Sequentially create all subscriptions for each component.
4) Sequentially create all publications for each component.
5) Enable the message declarations for each component to broadcast both the subscriptions and the publications that were created.

Depending on the OS of choice there can be significant throughput problems that arise from components that establish and declare a large volume of subscriptions and publications upon start up. These problems occur when a number of components create and declare subscriptions to the same message that other components are creating and declaring publications. A severe thrashing problem then occurs because each component is attempting to stay synchronized to the latest changes related to the message in common. Rather than each component being informed once of the subscribers and publishers of a message, each component that has already established a subscription or publication is notified

each time any other component creates and declares a subscription or publication to the message of interest. This results in a fragmented synchronization of subscriptions and publications between components. A more efficient solution is to have each component establish its subscriptions and publications in a wholesale manner that avoids possible thrashing.

Another problem that occurs, depending on the OS, is when a component establishes a subscription for a message and a different component establishes a publication to the same message after the first component has already declared the subscription. In this situation, there is sometimes a 30-45 time tic interval (dependent on OS/processor – as short as 100μs for VxWorks) that occurs because of considerations that are contained within the overriding message framework.

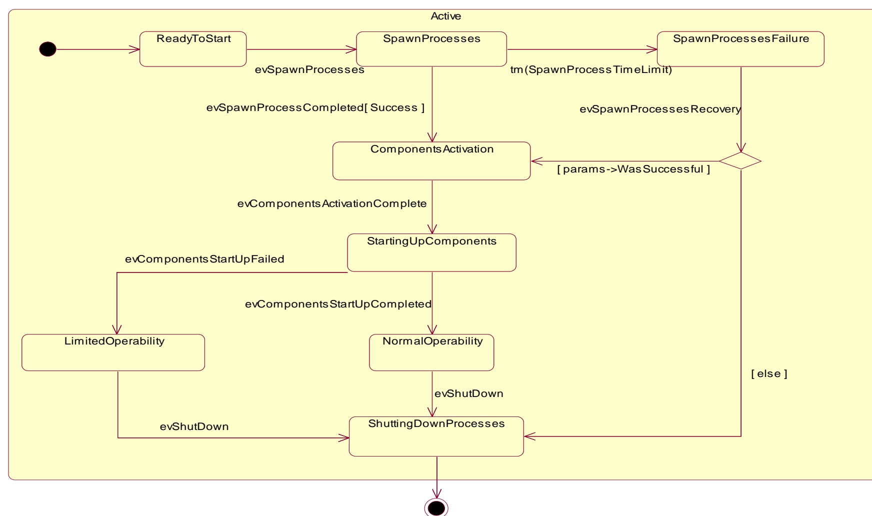

Figure 3
StartUpMasterSequence State Chart

The detail mechanics are presented in the state chart in Figure 3 above. The StartUpMaster sequences to each component in turn for each of these activities. Each activity is conducted for all components before the next activity is advanced to for any components. This ensures that no two components are creating subscriptions or publications, or broadcasting declarations for the same message at the same time or in an interleaved manner. In order to collaborate with the StartUpMaster to facilitate the sequencing required for subscriptions and publications, components can not create and declare subscriptions and publications in their constructors, but must instead expose operations dedicated to these functions that can be called by the StartUpMaster via the messages that they generate.

Figures 4 and 5 illustrate further the control mechanics in the appropriate state charts. The underlying state transitions highlight the overall processing control. In

the ComponentStarter State Chart each state is checked for Successful completion before transitioning to the next state or to a ComponentStartUpFailed state that handles the appropriate failures before routing back to start over again at event evBeginStartUp. The state transitions are handled the same way in Figure 5 for the ComponentActivation State Chart. All of these software components involve the system/subsystem object instantiations and reflect their functional performance in terms of message handling.

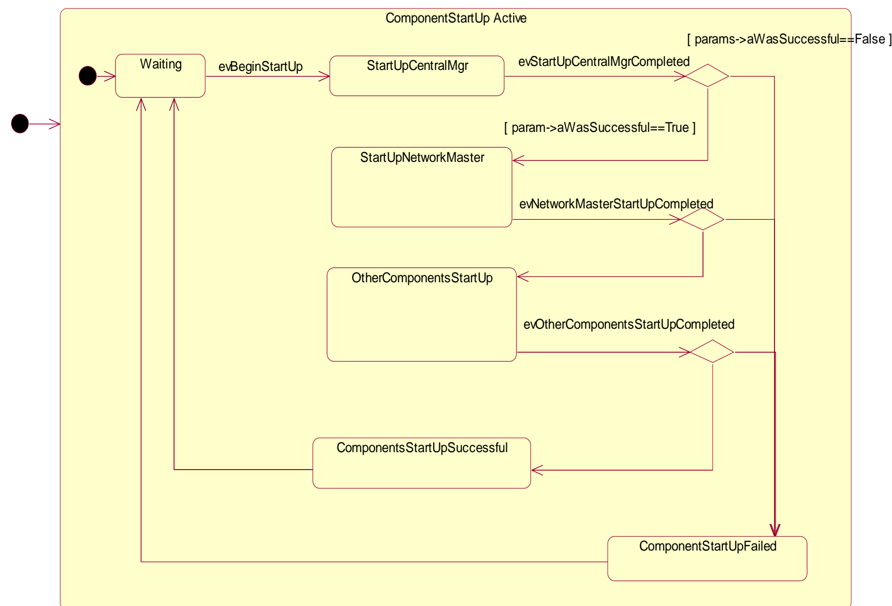

Figure 4
ComponentStarter State Chart

The StartUpMasterSequence imposes a protocol on software components that force allocation of memory for control classes (the more memory-persistent classes in the components) in the construction phase of the process. The memory for all memory-persistent classes in all components is allocated prior to the operation of any components. Before allowing any of the components to precede to operable modes the StartUpMasterSequence determines if all components were successful in memory allocation. Any shortage of memory is noted and constitutes a general system fault which should be addressed at the system level. Remember that we are using the term components to refer to the software modules that will represent the subsystem hardware components later on.

Figure 6 Concurrent ProcessSpawner State Chart shows more graphic details of the underlying control logic for state transitions that the StartUpMaster has kicked off. The underlying logic is detailed in this state chart for the set of protocols that are activated for each component development.

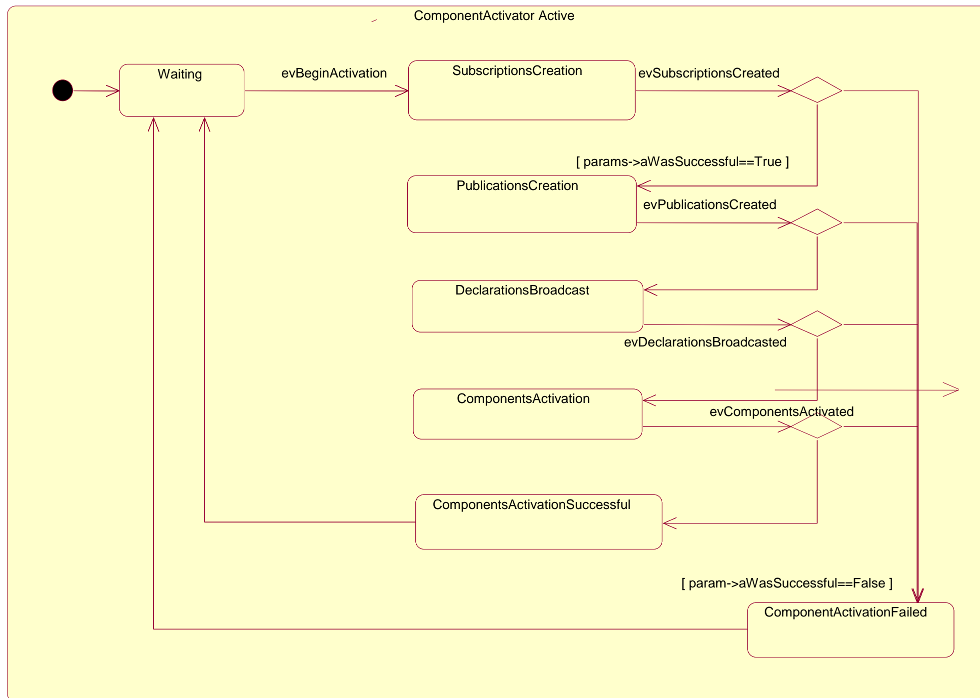

Figure 5
ComponentActivation State Chart

As in Figures 4 and 5 the framework is similar for handling success and failure at each state transition. In all of the diagrams above as failures occur and remain uncorrected a fault log can be maintained and the corresponding faults time stamped and reported.

2 Diagnostics Approach

All of these relationships are set up specifically to provide further interfaces and abstractions to allow for greater adaptability and applicability. The above processing allows a DiagnosticsManagerBuilder to be spawned that builds a singleton of the DiagnosticsManagerPkg.

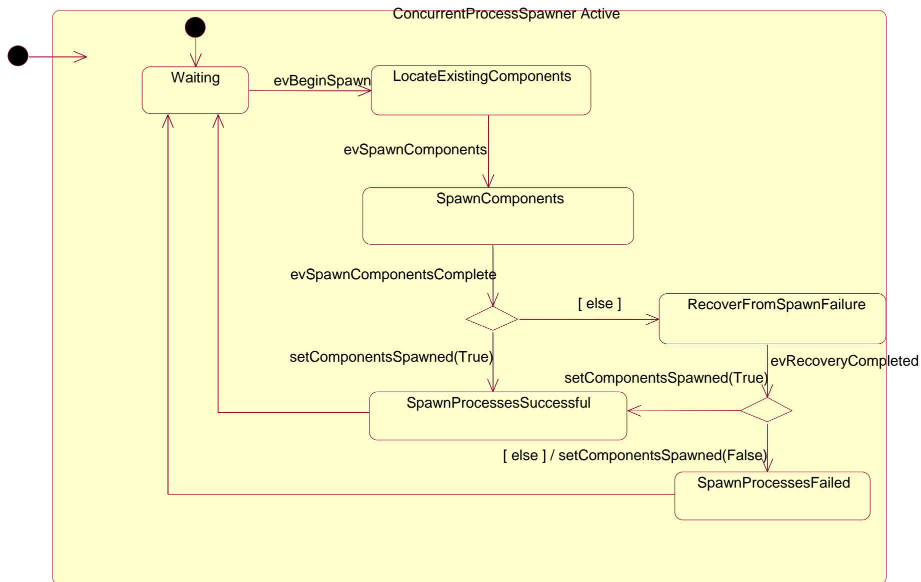

Figure 6

ConcurrentProcessSpawner State Chart

The class DiagnosticsManagerManager is initiated from the DiagnosticsManagerBuilder and Figure 7 DiagnosticsManagerManager State Chart illustrates the control logic mechanics that the class DiagnosticsManager inherits. The state charts featured in Figures 8 and 9 for the DiagnosticsManager present the logic. A succession of states is tested for success/failure at conditional connectors that further drives the data acquisition or acknowledges a failure.

In Figure 10 DiagnosticsObserver Pattern the class DiagnosticsManagerManager sets the overall process in motion for diagnostics and prognostics trending data acquisition and the DiagnosticsManager inherits the base features from this DiagnosticsManagerManager class in order to provide more defined state functionality.

In the DiagnosticsObserver Pattern the control process is established for data acquisition and fault monitoring. The DiagnosticsObserver acts as the inherited base class to all the control processes. This control process begins with the DiagnosticManager's inherited features from a DiagnosticManagerManager being coupled to states of possible modes (NormalOperability, ServiceOperability, ReducedOperability, DegradedOperability, and PrepareforShutDown). A DiagnosticsManagerMode class inherits these mode features in a generic Active state. Also, a sequence of derived classes from DiagnosticsManagerModes that act as filters inherits these mode features for further control.

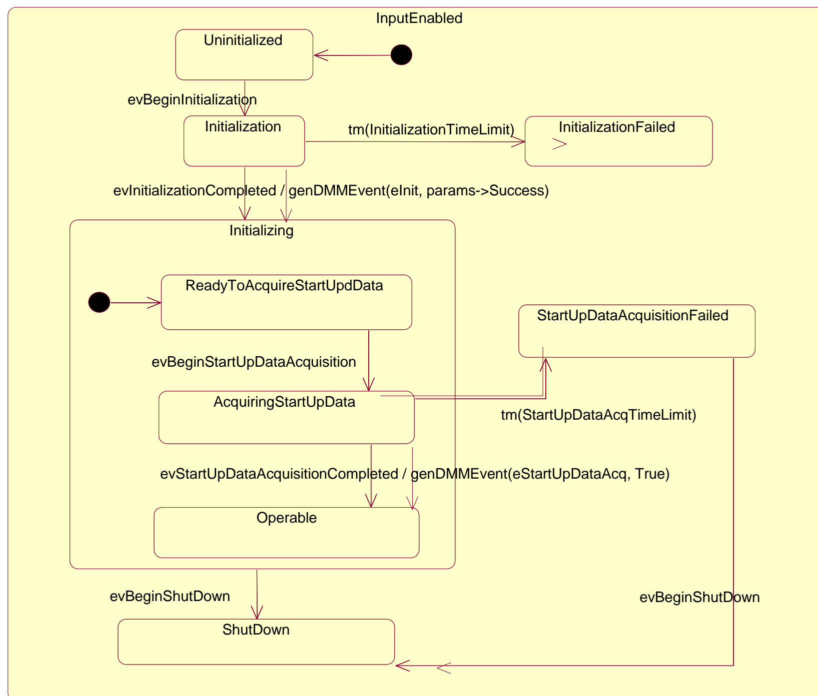

Figure 7
DiagnosticsManagerManager State Chart

Each of these derived classes such as InitializationMode; NormalMode etc. have their own set of specific overloaded/overridden functions that are originally defined as pure virtual functions in the AbstractDiagnosticsManagerMode and as virtual functions in the DiagnosticsManagerMode. These derived mode classes then provide specific state behavior for the mode operability. Each derived mode class controls the possible mode that the entire process can be in for the Maintenance Pattern as shown in Figure 11 and the Maintenance Pattern can only exist in the defined mode that each derived mode class represents.

This DiagnosticsManagerPkg provides the initial process controls to drive the state chart in the Maintenance Pattern as shown in Figure 11.

The **Observer** Design Pattern [1-3] is defined as follows:

– defines a one-to-many dependency between objects such that when one object changes state then all its dependents are notified and updated automatically.
– Publisher sends out notifications without having to know who the observers are. Any number observers can subscribe to receive notifications.
– provides abstract coupling and support for broadcast communication.

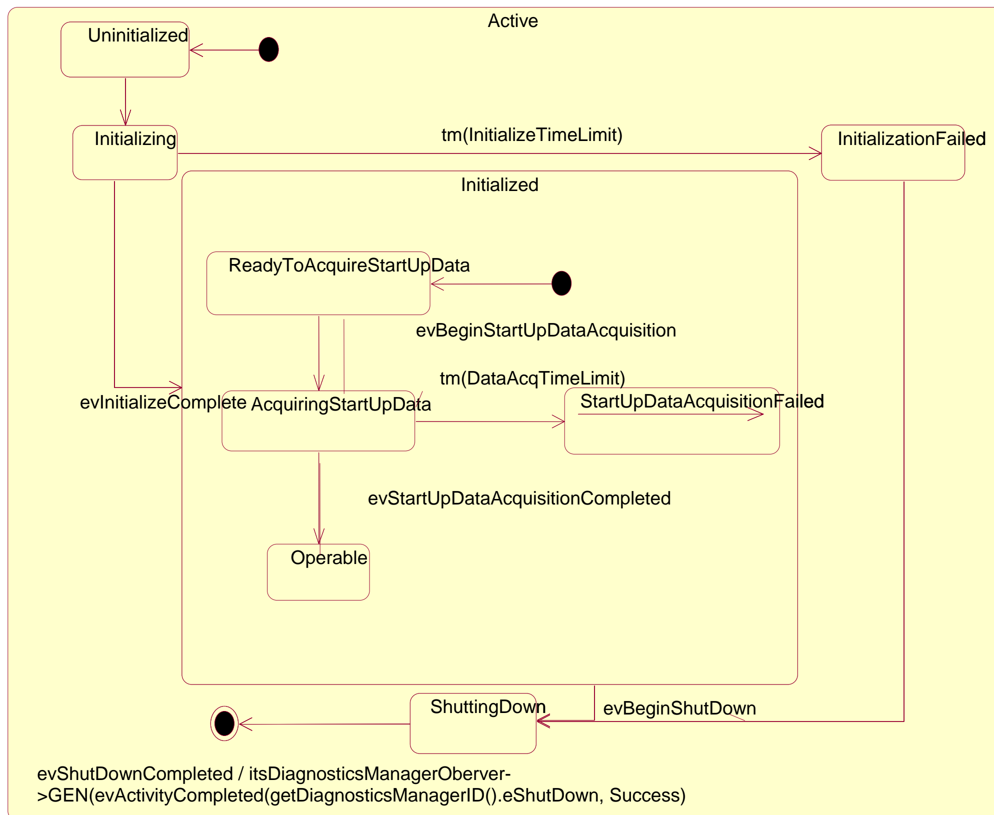

Figure 8
DiagnosticsManager State Chart

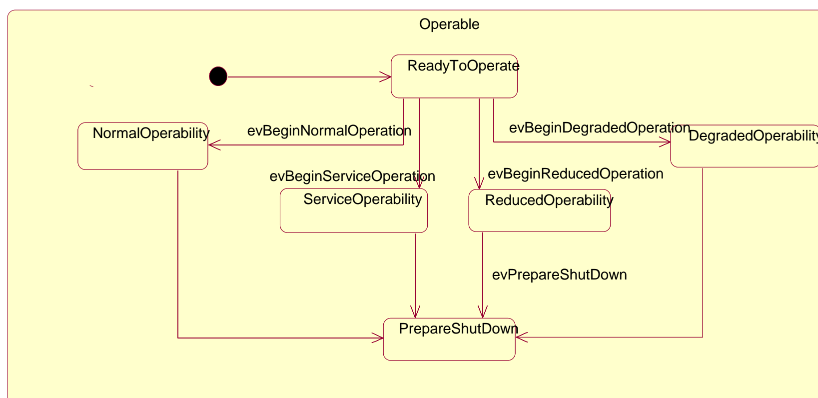

Figure 9
Operable State

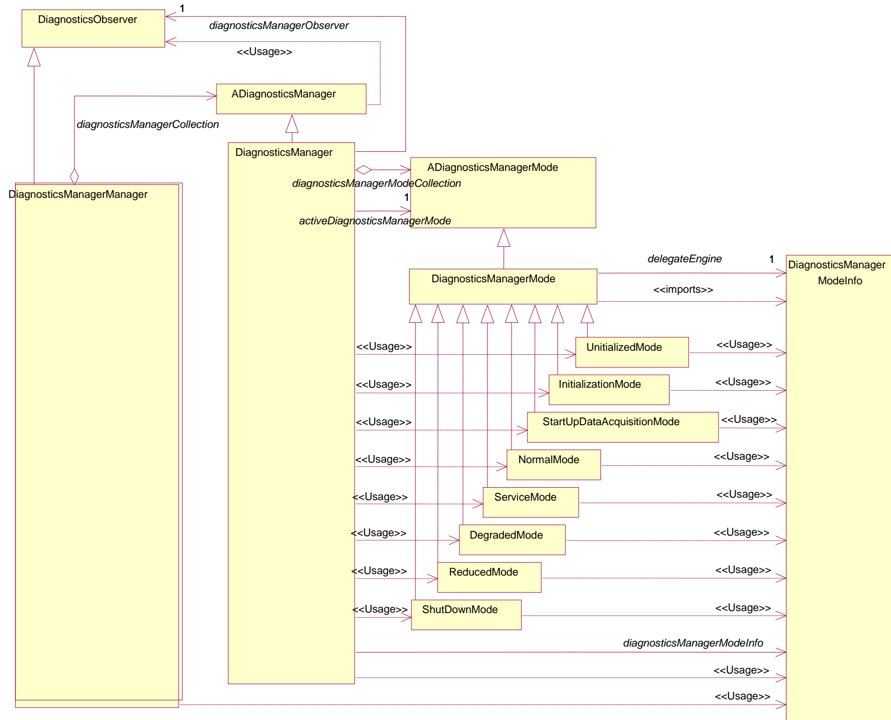

Figure 10
DiagnosticsObserver Pattern

In Figure 11 Maintenance Pattern a specialized Builder instantiates singletons for each class and starts all the state charts for the hardware subsystems OP1 etc. through the operation StartBehavior( ). This specialized Builder is an aggregate from the DiagnosticsBuilder-Broker (Figure 11 as continues on the right hand side of Figure 10 – actually being inside or connected to the interface for the DiagnosticsManagerModeInfo). A sequence of aggregations from the Builder to the AlgorithmManager class and then to the Algorithm class is established. The Algorithm class contains the functionality that is common to a set of subsystem classes OP1 etc that inherit from it and each subsystem customizes this functionality accordingly. For this functionality each subsystem can have multiple components deployed.

Upon instantiation the behavior of the Algorithm object enters a generic state chart as shown in Figure 12 States and Modes Design Pattern that every class such as Main and OP1 etc. enters upon instantiation and goes into an Active state. Each class such as OP1 etc. represents a subsystem. When StartConfig() begins on the Diagnostics level an event evStop() or triggered operation Stop() can be transmitted through the Diagnostics Manager interface to the AlgorithmManager and then to the inherited state charts from the Algorithm class to send all states

into a WaitToReset() state that acts as a holding state. This WaitToReset() state allows external access by the DiagnosticsObserver to all Diagnostics and Prognostics trending data as well as alarm logging. All the Processes can be halted in this state while Diagnostics proceeds.

With storing the configuration data and checking the alarms. The DiagnosticsObserver can then proceed to publish all of this data to Subscribers. An event evReset() or triggered operation Reset() can be transmitted to have all inherited state charts go into an initialization state that initializes all the data and alarm settings and then becomes a holding state until another event evActivate or triggered operation Activate() allows each process to restart. No objects are recreated so there is no reallocation of memory on the heap. All processes are simply suspended and restarted using the same local objects that were initially created upon spawning from the StartUpMaster to the DiagnosticsManager. The AlgorithmManager ensures that the processes stay in synchronization and all come up to restart at the same time.

Each subsystem as represented by classes OP1, OP2, OP3 etc. in Figure 11 has multiple components and Figure 12 States and Modes Design Pattern presents a generic sequence of states for data acquisition and fault monitoring (whereby each subsystem can be customized accordingly). Upon initial startup it is assumed that control data and attribute values are not established or reset from prior use and the entry point is the state Uninitialized. An event evInitialize allows the object to transition into an Initialization state for setting the preliminary values of control data and attributes. The event evActivate causes the object to transition into the main state of operation Active. Inside this state may be several nested states or modes that involve IntelligentAgents that monitor the system behavior and performance of the control devices. Faults as they occur are recorded and corrective action taken as necessary.

Inside Active the modes of operation are determined by the particular event. Upon an event evStop the object transitions out of Active into a WaitingToReset suspended buffer state to allow for data accumulation and avoid any possible data corruption. This suspended buffer state WaitingToReset allows external access by the DiagnosticsObserver to all Diagnostics fault and Prognostics trending data as well as alarm logs etc.

Either an event evReset or a ten minute interval tm(600000) from a reset timer allows the object to then transition to Initialization to begin the cycle of operation over again as necessary. There is inherent built in flexibility in these generic state charts for each subsystem found in the Maintenance Pattern. Each subsystem may require specific features such as timers, other events or no need for evReset or evStop etc. and can be defined locally. All that is required is the supporting features defined in the specialized Builder in Figure 11. The derived mode classes from Figure 10 that provide filtering for specific behavior could be embedded internally as nested state modal behavior inside the Active state chart of Figure 12.

The Figure 13 Intelligent Agent Mode illustrates the possible transitions inside an intelligent agent [4-5]. An agent is any control device that senses its environment that it is monitoring and acts upon that environment with actuators. It is deemed intelligent if it is capable of learning and adapting to its environment. An example is a fuel monitor with feedback control logic or artificial intelligence that enables it to learn and adapt to the environment it is monitoring as well as set alarms for faults as they occur. In the Intelligent Agent Mode the intelligent agent begins in a NoOp state until an event evMonitor allows the agent to transition into the activeOP state to monitor the environment. The agent continues to monitor its environment by means of a watchdog timer tm (100) that provides time intervals of 100ms.

An event evAccumulate causes the intelligent agent to transition into the state UpdateStatus where all diagnostics and prognostics trending data as well as acquired alarms are all accumulated and logged accordingly. The evStop causes the Active state to terminate as the object of interest exits out into the WaitingToReset suspended buffer state with all of the information acquired by the intelligent agent during this mode of operation. It is in this suspended buffer state WaitingToReset that all diagnostics and prognostics trending data and alarms logging is available for final accumulation and evaluation by the generic classes as well as the DiagnosticsObserver and can be updated to the Subscriber. IEEE Standards [6, 7] set the guidelines for how these agents should operate and report alarms. The entire set of subsystems OP1 etc. shown in the Maintenance Pattern of Figure 11 operate in a defined mode as dictated by Figure 10. Each subsystem could operate in different modes by building flexibility into the Intelligent Agent Mode of Figure 13. These design decisions need to be decided upon if all the subsystems transition together to report faults periodically or each subsystem transitions independently to report faults as necessary. Figures 10 and 11 illustrate how all subsystems transition together to report faults periodically. These figures also detail the flexibility that could also be made available in the States and Modes Design Pattern of Figure 12.

In Figure 14 Pattern of Patterns the DiagnosticsBuilder initializes all classes and state charts and enables the state charts. For a network centric platform this Pattern of Patterns approach offers an ideal way of initializing data and associations through the DiagnosticsBuilder. The CORBA ORBS that the Diagnostics Broker could represent provide locking and unlocking of object references such as instantiated component subsystems that are manufactured by the DiagnosticsFactory. These instantiated objects are then utilized by the DiagnosticsObserver in subsequent state charts and modes of operation as depicted in Figures 10 and 11 above. The state chart behavior these objects enter into is depicted in Figures 12 and 13.

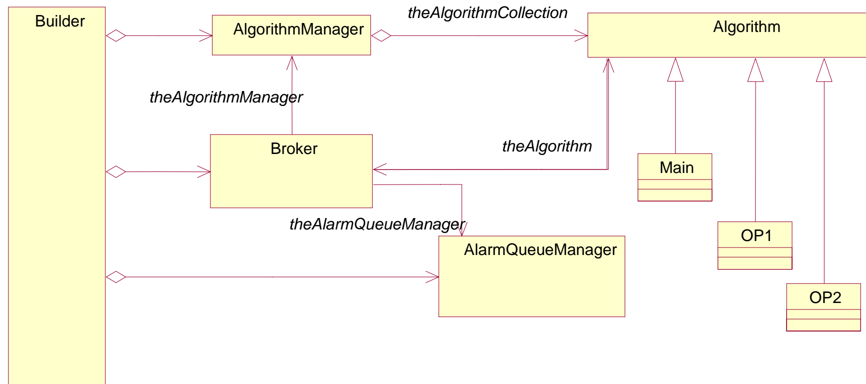

Figure 11
Maintenance Pattern

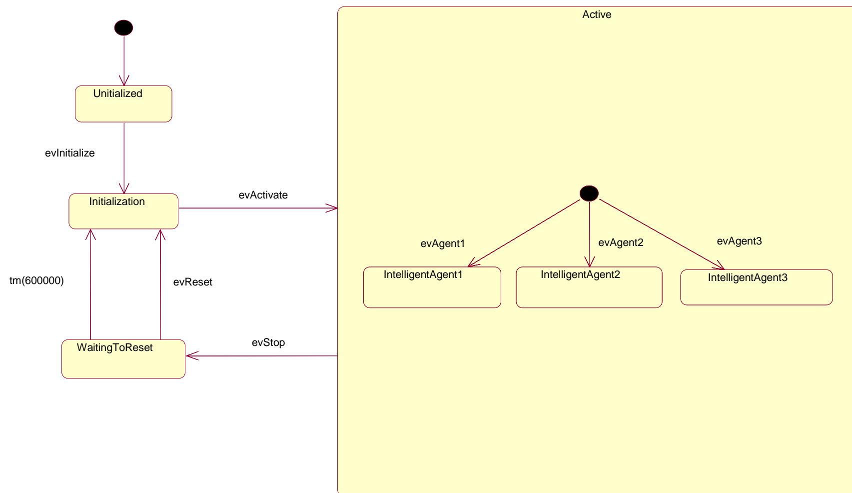

Figure 12
STATES and MODES DESIGN PATTERN

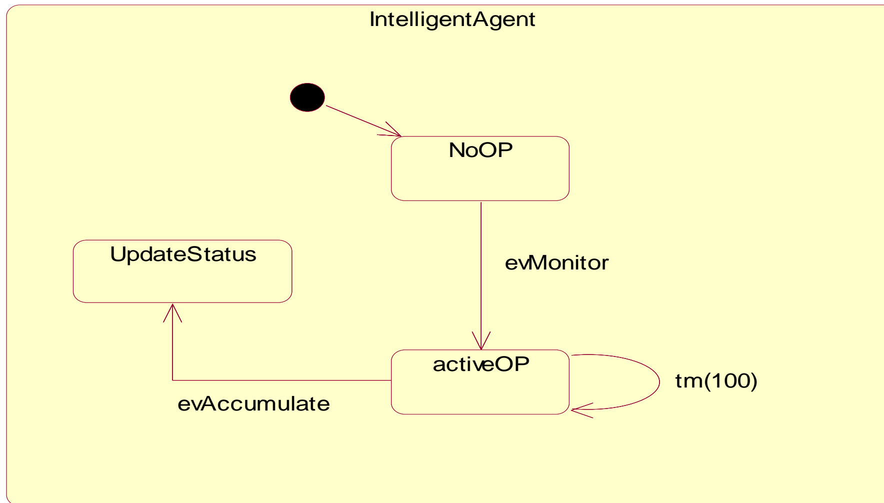

Figure 13
INTELLIGENT AGENT MODE

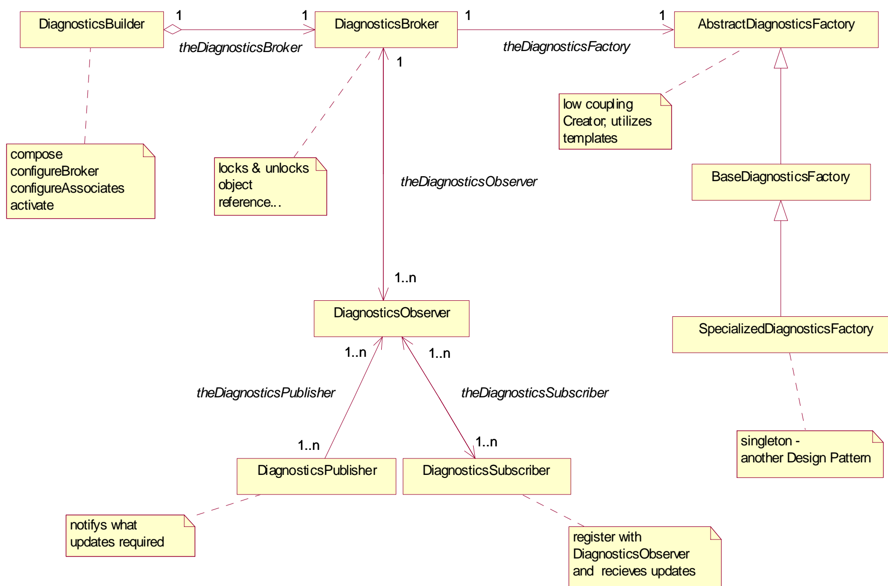

Figure 14
Pattern of Patterns

In the above Figure the DiagnosticsBroker behaves as an elaborated proxy between client and server; for example the ORBs in CORBA are essentially a **Broker** Design Pattern [1-3]. Encapsulating both data and operations the ORBs provide interfacial transmission across the network. The overall behavior of this design pattern is as follows:

- decouples clients from server. Object Broker knows the location of other objects.
- can construct a Proxy Pattern when location of the server is not known at compile time.
- **CORBA**
  - acting as an object reference repository:
    - Dynamic – mediates all requests so no linkage between client and server
    - Static – reference address for server, client uses to connect directly with server side proxy. Server objects register with Broker.
  - coordinates communication to transmit results & exceptions.

Another design pattern that is most utilized in the Pattern of Patterns approach is the DiagnosticsFactory which is a **Factory** Design Pattern [1-3]. Its behavior is described as:

- virtual constructor as defining an interface for object creation but lets subclasses decide which class to instantiate. Allows more freedom of abstraction (**low coupling**).

  Class templates are an excellent example of this.
  - manufactures an object especially when it cannot be anticipated the class of objects to create.
  - connects parallel class hierarchies. These result when a class delegates some of
  - its responsibilities to separate classes.

The DiagnosticsObserver in Figure 14 Pattern of Patterns has access to everything in its own defined pattern domain as presented in Figure 10. This DiagnosticsObserver Pattern is a top level diagram to Figure 11 Maintenance Pattern which becomes interfaced through the class under usage from DiagnosticsManagerModeInfo. For large systems with numerous subsystems this architectural hierarchy of patterns provides an automated, standardized and validated method of providing real time diagnostics and prognostics capability provided that a powerful processor OS is used with large memory capacity and speed.

Conclusion

This paper presented a series of UML class and state chart diagrams that illustrated the mechanics of having a reliable and coordinated startup sequence with all components. In addition several state charts showed a proposed approach for diagnostics fault handling and fault trending for prognostics. Intelligent agents were introduced as smart control devices that could act on the environment they are monitoring. A Diagnostics Fault handling class diagram was discussed that showed how all components could be powered up and the respective class relationships. A Diagnostics Maintenance pattern illustrated the use of inheritance with an algorithm manager handling different operations OPS. Finally a Pattern of Patterns design pattern was proposed that illustrated the superposition of all other patterns utilized in fault handling – Factory for singletons, Builder for accumulating classes, Broker for distributed communications using CORBA ORBS and an Observer utilizing publication and subscription features.

References

[1] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns, Elements of Reusable Object-Oriented Software, Upper Saddle River, NJ: Addison Wesley, 1995, 97-135

[2] Douglass, B.: Doing Hard Time, Developing Real-Time Systems with UML, Objects, Frameworks and Patterns, Reading, MA: Addison Wesley Longman, 2000, 434-465

[3] Larman C.: Applying UML and Patterns, An Introduction to Object-Oriented Analysis Design and the Unified Process, Upper Saddle River, NJ: Prentice Hall, 2nd Ed., 2002, 449-50

[4] Russell S., Norvig P.: Artificial Intelligence, A Modern Approach, 2nd Ed., Upper Saddle River, NJ: Prentice Hall 2003, Ch. 2, Intelligent Agents

[5] Fault Diagnostics/Prognostics for Equipment Reliability and Health Maintenance, Seminar by Georgia Tech for Distance Learning and Professional Education, May 18-21, 2004, Atlanta, Ga., Section VII PHM and CBM

[6] IEEE P1522/D3 'IEEE Draft Trial Use Standard for Testability and Diagnosability Characteristics and Metrics', IEEE Standards Coordinating Committee 20 on Test and Diagnosis for Electronic Systems, May 2004

[7] IEEE 1232 'IEEE Standard for Artificial Intelligence Exchange and Service Tie to All Test Environments (AI-ESTATE)', IEEE Standards Coordinating Committee, November 20-22, 2002