Architectural Design Patterns Customized and Validated for Flight Software

U.V.R. Sarma[1], N. Pavani[2] and Dr. P. Premchand[3]
[1] CVR College of Engineering, Department of CSE, Ibrahimpatan, R.R. District, A.P., India
Email: sarmauvr@yahoo.co.in
[2] CVR College of Engineering, Department of CSE, Ibrahimpatan, R.R. District, A.P., India
Email: mee_pav@yahoo.com
[3]Osmania University, Department of CSE, Hyderabad, A.P., India
Email: p.premchand@uceou.edu

Abstract— **Software design patterns are best practice solutions to common software design problems. When they are properly applied, software design patterns can greatly improve the quality of software architectures. Leveraging the benefits of design patterns is particularly important in the space Flight Software (FSW) domain because better designs are needed to help reduce the number of flight software related anomalies and thus improve the quality of FSW architectures. This paper provides a solution to build templates for common features of Flight Software architecture using design patterns. This is illustrated by using Student Nitric Oxide Explorer (SNOE) spacecraft, which is a real world case study from National Aeronautics and Space Administration (NASA). The executable design pattern templates help an engineer when building software architectures. This paper also provides a foundation to perform validation for functional correctness during the design phase.**

Key Words: **Software Architectural Design Patterns, UML, Space Flight Software, IBM Rational Rhapsody.**

I. INTRODUCTION

Software design patterns are best practice solutions to common software problems. Design patterns are normally captured to be domain and platform independent. There are several benefits of capturing design patterns in this manner. First, it makes them applicable across multiple domains and platforms. Second, it makes design patterns applicable at different levels of abstraction. Furthermore, in the majority of cases, multiple design patterns can be applied in a single application.

To achieve this goal, this paper provides a set of design patterns that are applicable to a small satellite Student Nitric Oxide Explorer (SNOE). This paper also describes a validation approach that is used to validate the functionality of software architectures.

This paper is applied and validated using the space Flight Software (FSW) domain. FSW is an ideal domain to apply this dissertation for multiple reasons. First, the amount of requirements and responsibilities placed on FSW is growing. FSW has evolved from performing simple operations to controlling a majority of the spacecraft payloads. This paper is a way to architect FSW using design patterns. Using design patterns makes certain that best practices are incorporated into FSW designs.

Secondly, the industry trend indicates that the number of software related anomalies is growing. It is reported that "in the period from 1998 to 2000, nearly half of all observed spacecraft anomalies were related to software" [1]. These software anomalies can cause mission disruption or even mission loss. In the aerospace industry these losses cannot be tolerated because of the high cost and length of time that is required to build a spacecraft. Additionally, many spacecrafts support very critical missions that can be severely impacted from a small disruption of service. This paper helps to alleviate the number of software related anomalies by providing design time validation. Therefore, design flaws that lead to software anomalies can be identified and remedied early.

This paper is organized as follows. First it describes about IBM Rational Rhapsody. Next, about UML 2.0 and how it was used in the paper, then about SNOE and the process for customizing general design patterns for SNOE using IBM Rational Rhapsody is described in detail. Finally, this paper includes a discussion on conclusions and areas of future work.

II. IBM RATIONAL RHAPSODY

This project uses IBM Rational Rhapsody to build and execute the state machines [2]. Therefore the actions performed are captured using IBM Rational Rhapsody's action language and event handling infrastructure. IBM Rational Rhapsody uses custom action language, which is a subset of the Java language, to capture actions and to execute the model. Thus, this action language is used to implement the objects actions. The action language is similar to Java, except there are a few additional reserved words and functions. For example, GEN is a reserved word used to generate asynchronous messages as events. The messages must be specified on the consumer's provided interface in order to be invoked.

Ex: PClass1.gen(new msg());

Where PClass1 is the provided interface which also specifies the port through which the message is sent and msg() is event that is generated. When an event is generated, IBM Rational Rhapsody event handling infrastructure handles the routing of events from the producer to the consumer. When the consumer component receives the event, the appropriate state transition is taken and actions

DOI: 10.32377/cvrjst0304

within that state are performed. IBM Rational Rhapsody is an excellent tool to generate dynamic UML diagrams using Real-time UML that is UML 2.0.

III. UML 2.0 As Architectural Description Language (ADL)

The Unified Modeling Language (UML) [3] is formal graphical language considered as a de facto industrial standard. Although the language has been created as graphical language firstly to support object oriented software analysis and design, the language has been revised couple of times and today, it is a general formal language capable to describe a software system. The UML has well defined formal syntax and semantics and can be machine checked and processed. UML includes a set of graphical notation techniques to create abstract models of specific systems.

The expressive power of Architectures by UML is more than any ADL. The UML profile for scheduling, performance, and time specification described in [4] has been adopted as an official OMG standard in March 2002. In general, UML profile defines a domain specific interpretation of UML; it might be viewed as a package of specializations of general UML concepts that capture domain-specific variations and usage patterns. To specify a profile, UML extensibility mechanisms (i.e., stereotypes, tagged values, constraints) are used.

Component and connector views (C&C views, for short) present an architecture in terms of elements that have a runtime presence (e.g., processes, clients, and data stores) and pathways of interaction (e.g., communication links and protocols, information flows, and access to shared resources). *Components* are the principal units of run-time interaction or data storage. C*onnectors* are the interaction mechanisms among components.

The components are created as Composite classes in UML 2.0 and each of the components should have ports to interact with the external environment. Each port again requires an interface for it to interact. The interfaces are of two types *Provided Interface* and *Required Interface.* Two components with ports and their interfaces can be linked for communication. The ports and their interfaces should be compatible, that is one component having a required interface (depicted as semi circle) can interact with only a component that provides the interface (depicted as full circle).

IV. Student Nitric Oxide Explorer (SNOE)

This paper illustrates the construction of architecture for Flight Software by taking up a case study of Student Nitric Oxide Explorer (SNOE) [5]. SNOE, which was a real-world, small satellite program funded by the National Aeronautics and Space Administration (NASA) and managed by the Universities Space Research Association (USRA).

SNOE's mission involves using a spin stabilized spacecraft in a low earth orbit to measure thermospheric Nitric Oxide (NO) and its variability. The SNOE spacecraft is spin stabilized, meaning it maintains its orientation similar to that of a top. SNOE is required to maintain a spin rate of 5 Rotations per Minute (RPM). The spin rate can be adjusted having the Flight Software (FSW) send a command to commutate the electromagnet transverse torque rod. The spin axis direction is controlled in a similar fashion by having the FSW send a command to commutate the electromagnet spin axis torque rod. SNOE's FSW does not perform the attitude determination and control calculations. Rather, the FSW collects the attitude measurements and downlinks them to the ground for processing.

Then the ground uplinks attitude control commands back to the spacecraft for the SNOE FSW to execute. The attitude measurements are taken from two Horizon Crossing Indicators (HCI) and three magnetometers. SNOE's spacecraft body is surrounded on all sides by stationary solar panels which are used to generate power.

The spacecraft contains three payload instruments to accomplish its scientific mission. These three instruments are an Ultra Violet Spectrometer (UVS) that measures NO density, an Auroral Photometer (AP) that measures the flux of energetic electrons entering the Earth's upper atmosphere, and a Solar soft X-ray Photometer (SXP) that measures the solar irradiance.

Additionally, SNOE also contains a microGPS Bit-Grabber Space Receiver (microGPS BGSR) instrument as a technology experiment. The microGPS BGSR gathers position information based on the Global Positioning System (GPS) constellation for experimental orbital determination.

4.1 SNOE Design Pattern Selection

The pattern selection process is done using the command execution functionality, which is a commonly seen in FSW. This involves determining the order in which spacecraft commands are executed. The design patterns that support this feature are then selected. For example, on small spacecraft the centralized control design pattern [6] can be used. The centralized control design pattern involves a single controller that provides overall control by conceptually executing a state machine. This design pattern is useful on small spacecraft because it encapsulates all the state-dependent control in a single component thus making the control logic easier to understand and maintain. Thus, the design patterns that support SNOE specific features are determined by selecting the Design Patterns that are suitable for the working of SNOE.

The paper illustrates the customization of Design Patterns to suit the architecture of the satellite Student Nitric Oxide Explorer (SNOE). Seven different Design Patterns have been identified to reflect the functionality of SNOE.

The Design Patterns identified are listed in Table I.

Table I. SNOE Design Pattern Selection

Feature	Design Pattern
Command Execution	Centralized Control Design Pattern
Telemetry Storage and Retrieval	Telemetry Client Server Design Pattern
Telemetry Formation	Pipes and Filters Design Pattern
Ground Driven Payload Data Collection	Payload Multiple Client Multiple Server Design Pattern
Ground Driven Housekeeping Data Collection	Housekeeping Multiple Client Multiple Server Design Pattern
Spacecraft Clock	Spacecraft Clock Multicast Design Pattern
Memory Storage Device Fault Detection	Memory Storage Device Watchdog Design Pattern

The reason for selecting the above Design patterns is described below:

4.1.1. Centralized Control Design Pattern: SNOE is a small satellite with thirteen different components. Since it is a small satellite and the number of components is less, Centralized control architecture is better suitable than Distributed architecture. The Centralized controller is linked to every component and controls the functionalities of each of the components.

4.1.2. Telemetry Client Server Design Pattern: The information collected by various components in SNOE is transformed into telemetry packets and is sent to the Ground Station. Every component has its own buffer and stores the information collected by them in their buffers. Next, the information is to be periodically transformed into telemetry packets and is to be sent to the Ground Station for processing. So a Client and Server component is created for each of the components which will be controlled by the Centralized Controller and will be responsible to collect the information. This pattern collects information from Payload Server as well as HouseKeeping Server and sends it to the controller.

4.1.3. Telemetry Formation Pipes and Filters Design Pattern: The transformation of information into telemetry packets is done by Pipes and Filters Design Pattern. It increases throughput capacity of the system by adding multiple homogeneous (identical) channels.

4.1.4. Payload Multiple Client Multiple Server Design Pattern: There are four payload instruments in SNOE. They are Ultra Violet Spectrometer, Micro GPS, Solar XRay Photometer and Auroral Photometer. A separate client and server for each of the payload instruments are created to collect the information whenever the controller signals to collect.

4.1.5. Housekeeping Multiple Client Multiple Server Design Pattern: The health of the satellite is maintained by collecting the information of the health or working of each of the component. This information is sent to the ground station. The ground station checks this information and sends any signals if necessary to check and modify the components. The collection of housekeeping information is done by this Design Pattern. Again a separate client and server component is created for 13 components of SNOE.

4.1.6. Spacecraft Clock Multicast Design Pattern: This pattern is used to send time signals to the Centralized controller and input and output components of the system.

4.1.7. Memory Storage Device Watchdog Design Pattern: The memory storage device in SNOE is EEPROM. The Memory Storage Watchdog Design Pattern is selected to check the working of the memory storage device that is the EEPROM at regular intervals.

V. Implementation

5.1 *SNOE Centralized Control Architectural Design Pattern*

SNOE utilizes the Centralized Control design pattern to execute commands and control the overall operation of the spacecraft. SNOE uses two torque rods, thus its multiplicity is one or many. Additionally, the ports and interfaces for the payload variants that are unique to SNOE are modeled. The component diagram for SNOE's Centralized Control component diagram is shown in fig. 2, which contains the SNOE specific variants based on SNOE's features. The ports, interfaces, and connecters for the common variants are captured in the diagram.

SNOE contains four payload devices - therefore four payload device variants are created. For each payload variant, the port name is updated to reflect the specific payload, such as the *microGPS_IOC*. The port's interface is updated to reflect the specific actions that can be invoked on that payload.

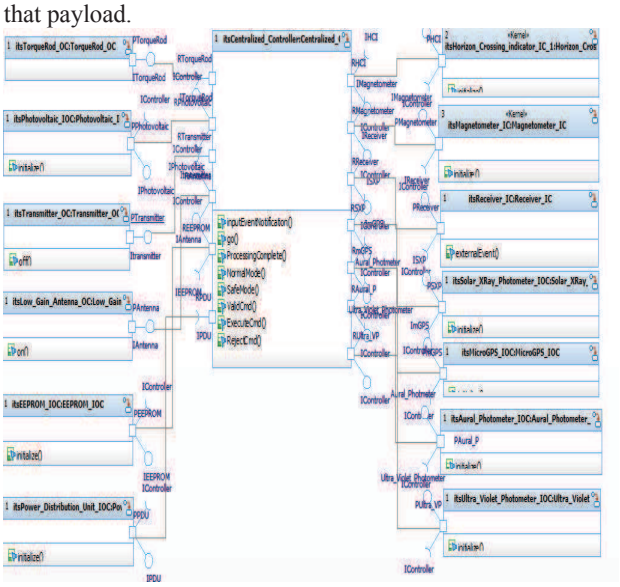

Figure 2. Component diagram for SNOE Centralized Control Executable Design Pattern

Next, the executable version of the design pattern involves potentially adding application specific states,

DOI: 10.32377/cvrjst0304

actions, and activities to the state machines based on the application's features. For example, if the application features refine some behavior, then this can be modeled as sub-states. Also, if the component must send a message to an application specific variant or if application specific logic is required then this is modeled as an action or activity within a state or transition.

SNOE receives command to open the solar x-ray photometer door; it knows the precise operations to invoke on the Solar_Xray_Photometer_IOC. The state machine for the Solar_Xray_Photometer_IOC component is depicted in the fig. 3. The state machine for this Component is slightly more complicated because it acts as both an input and IO component. The component begins in the Idle state within the Working state. In the Idle state the Component waits for commands from the Centralized_Controller. When an action message is received, it transitions into the Executing_Command state where it performs the appropriate actions on the external hardware. After it performs the necessary actions, it generates the processingComplete event and transitions back to the Idle state to wait for the next command. When a read message is received, a similar set of states and transitions occurs, however, it occurs in the Gathering Data state. The IO_Component is also responsible for listening to external events from the hardware. Therefore if an externalEvent event is received, the IO_Component stops its current action in the Working state and transitions into the Preparing_Notification state. In the Preparing_Notification state it prepares a message to send to the Centralized_Controller.

Once the message is ready, the IO_Component then sends the inputEventNotification message to the Centralized_Controller through the RIO port and transitions back to its previously interrupted location within the Working state.

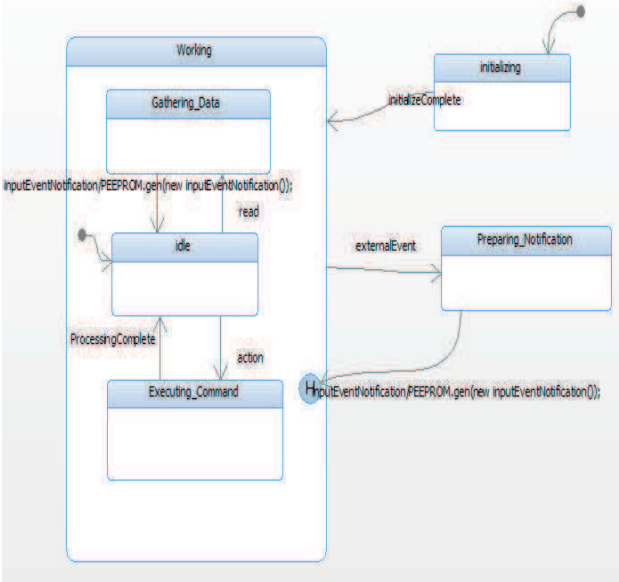

Figure 3. State Machine for Solar_XRay_Photometer_IOC

Next the state machine for the SNOE's Magnetometer_IC component is depicted in Figure 4. Magnetometer is an input component that provides attitude measurements. It is initialized by the Centralized Controller. It is first in the idle state and moves to the Preparing_Notification state when an external event occurs. Here it prepares the input_event_notification and sends it to the Centralized_Controller. A similar set of actions is performed in response to a read event message; however the requested data is collected and sent back the Centralized_Controller.

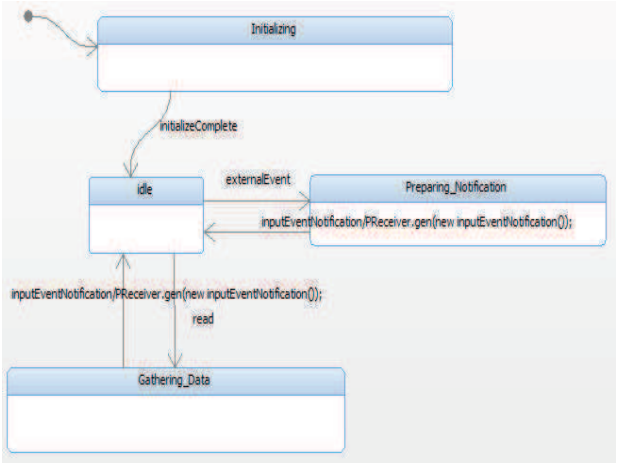

Figure 4. State Machine for Magnetometer_IC

The Output Component begins in the Idle state where is waits for commands from the Centralized_Controller. Once a command message is received, the Output_Component transitions into the Executing_Command state where it performs the appropriate actions on the external hardware. At the DRE level, the actions the Output_Component performs are variable; therefore it is modeled using a code stub as seen in fig. 5. Once complete, it generates the processing Complete event and transitions back to the Idle state to wait for the next command.

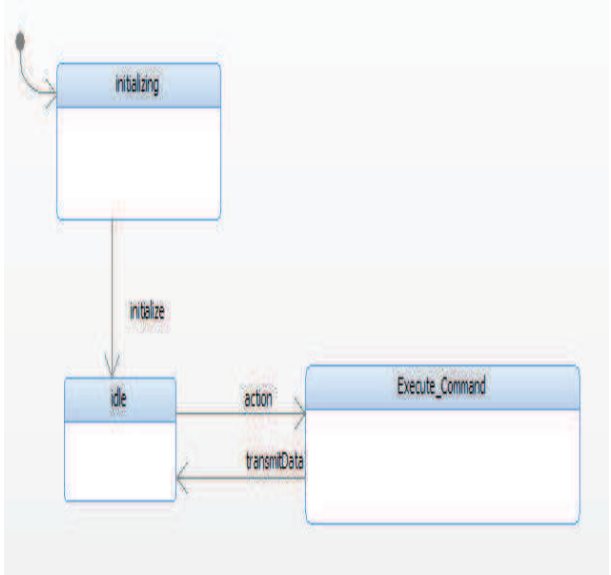

Figure 5. State Machine for Low_Gain_Antenna_OC

Finally, the state machines for the other variant input, output, and IO components, are also added.

5.2 SNOE Payload Multiple Client Multiple Server Executable Architectural Design Pattern

The next executable design pattern realized in SNOE is the FSW Payload Multiple Client Multiple Server executable design pattern. This design pattern is used to selectively collect payload data. Since SNOE is required to selectively collect the payload data, separate data clients are created for each payload instrument. Additionally, since each payload instrument has its own data buffer, separate server components are created for each payload instrument.

Next, the component diagram in fig. 6 depicts the set of components in the system. The ports and connectors added between the appropriate clients and servers are also shown in the diagram. Additionally, the interfaces are also updated to reflect the SNOE's unique variants. The diagram shows that the connected components have compatible interfaces.

The four Payload instruments include Ultra_Violet_Spectrometer, Micro_GPS, Solar_XRay_Photometer and Aural_Photometer. A client and server component for each of the four payload instruments is depicted in the design pattern.

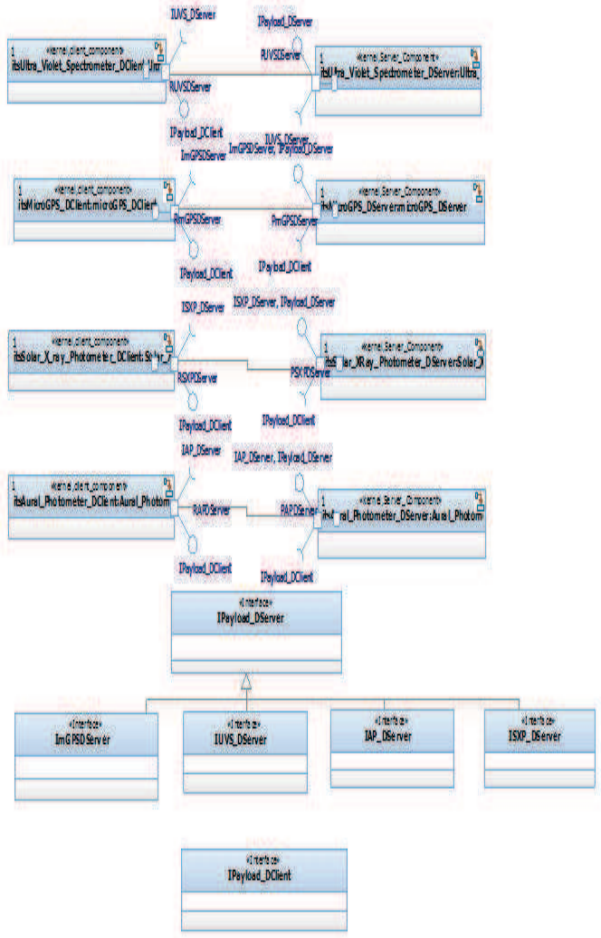

Figure 6. Object Model Diagram for Payload Multiple Client Multiple Server

The SNOE Multiple Client Multiple Server design pattern involves selectively collecting payload data. The interaction diagram for collecting micro GPS (Global Positioning System) data is depicted in Figure 7.

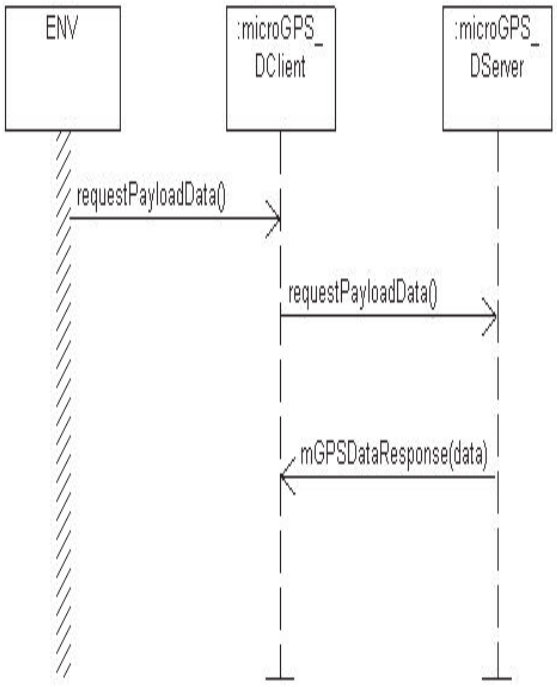

Figure 7. Collect microGPS Data Scenario for SNOE

5.3 SNOE Payload Multiple Client Multiple Server Executable Design Pattern

In addition to updating the architectural views, the executable version of the design pattern also needs to be customized for SNOE. This is performed for each client and server in this design pattern. The specific steps involved in updating the state machine are follows.

First, the microGPS_DClient component is responsible for collecting the microGPS data from the microGPS_DServer. The state machine for the SNOE specific microGPS_DClient component is depicted in fig. 8.

When Controller requires data it sends requestPayloadDataNeeded message to microGPS_DClient. microGPS_DClient requests the data from the server, this information is added to the actions on the state machine. This information is captured on the transition from the Preparing_Request state to the Idle state. The event that occurs is the requestPayloadData and the action

RUVSDServer.gen(newrequestPayloadData(msg));

Indicates that a request for payload data is being sent to the microGPS_DServer component by specifying the required port (RmGPSDServer) of the client through which the components communicate. Finally, the SNOE specific processing logic within the Preparing_Request and Processing_Response states is added as On Entry actions. However, this information is not depicted in Figure in an effort to make the diagram readable.

DOI: 10.32377/cvrjst0304

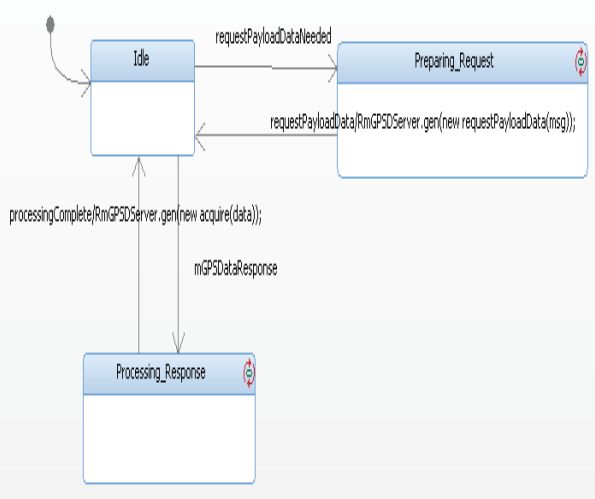

Figure 8, State Machine for MicroGPS Client

The state machine for microGPS_DServer in Figure 9. depicts the transitions that server takes. It is in Idle state first and moves to Processing_Client_Request state when client sends a requestPayloadData to server. After processing is complete, it prepares a response and moves back to the Idle state. During this transition it sends the mGPSDataResponse back to the client.

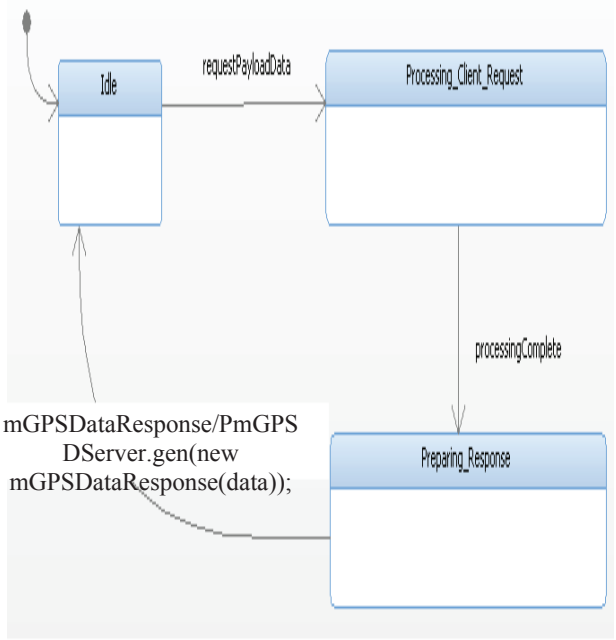

Figure 9. microGPS_DServer state machine

Similarly, the state machines for client and server for the other three payload instruments which are Auroral_Photometer, Solar_Xray_Photometer and Ultraviolet_Spectrometer are also updated following a similar process.

CONCLUSIONS

This paper describes an approach for building FSW software architectures from software architectural patterns. This approach improves the quality of FSW software architectures because it leverages best practices captured in software design patterns. Additionally, the executable design pattern templates not only help an engineer when building software architectures, but they also provide the foundation for performing design time validation on the software architecture produced using this approach. The engineers also can use the design patterns to form the core base for building the software architecture of any other system in this domain. Thus enabling to develop using the Software Product Line (SPL) based product development.

FUTURE ENHANCEMENTS

There are several avenues of future research that can be taken to extend this project. First, the SNOE case study can be expanded to include performance validation using MARTE (Modeling and Analysis of Real-Time Embedded systems) stereotypes. Second, this work should be extended to address feature modeling to help organize and structure the functionality of design patterns. Thirdly, this work can be applied to other DRE domains to illustrate the approach's applicability across DRE domains. Additionally, future research can include illustrating the functionality of the design patterns using the "animation" feature of IBM Rational Rhapsody.

REFERENCES

[1] Julie Street Fant, Hassan Gomaa, Robert G. Pettit, *Architectural Design Patterns for Flight Software*, 14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops, 2011.

[2] D. Harel, *Executable object modeling with statecharts*, *18th International Conference on Software Engineering*, 1997.

[3] B.Bharathi, Dr.D.Sridharan, *UML as an Architecture Description Language,* International Journal of Recent Trends in Engineering, Vol. 1, No. 2, May 2009

[4] Clements. P. et.al.: *Documenting Software Architectures, Views and Beyond*, Addison-Wesley, Boston, MA, USA,2002.

[5] Laboratory For Atmospheric and Space Physics at the University of Colorado at Boulder, *Student Nitric Oxide Explorer Homepage*, *http://lasp.colorado.edu/snoe/*. [Online]. Available: http://lasp.colorado.edu/snoe/. [Accessed: 21-Apr-2010].

[6] H. Gomaa, *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures,* Addison-Wesley Object Technology Series,, 2005.