Building Software Architecture using Architectural Design Patterns

U.V.R. Sarma
Department of CSE
CVR College of Engg.
Ibrahimpatan(M),
R.R. District, A.P., India

Neelakantam Pavani
Department of IT
CVR College of Engg.
Ibrahimpatnam(M)
R.R. District, A.P., India

P. Premchand
Osmania University,
Department of CSE,
Hyderabad, A.P., India

Abstract: this paper discusses how Software Architectural design patterns could be used to build the architecture of a system. The application of design patterns helps to improve the quality of software architecture and to reduce the flaws in the architecture. Generic architectural design patterns for real-time software components are customized to suit the functionality of system. This is illustrated using the Solar TErrestrial RElations Observatory (STEREO) a case study based on NASA’s STEREO mission. The customized design patterns are validated using IBM Rational Rhapsody. These validated design patterns form templates for further use in building the architecture of flight software.

Keywords: Software Architectural Design Patterns, UML 2.0, IBM Rational Rhapsody Developer for Java, Flight Software, Components.

1. INTRODUCTION

This paper customizes generic design patterns to suit the functionality of Solar Terrestrial Relations Observatory (STEREO). The Design Patterns are built and also validated using IBM Rational Rhapsody. The variability in the design patterns can also be represented in the diagrams by using Product Line UML based Software Engineering methodology by Gomaa [1] to enable the use of design patterns for any other systems in DRE domain.

The STEREO case study is chosen as the number of flight software anomalies are increasing in number and also as they lead to major losses [2]. Thus as per the functionality of STEREO ten generic design patterns are identified and are customized to suit the functionality of STEREO. The functionality of each component of STEREO is depicted by using state charts. The paper uses the IBM Rational Rhapsody developer for Java 7.6.1 to build the design patterns and also to build and validate the state charts. Validating the design patterns and state charts will better describe the functionality of each component of the system and will check the design patterns for their functional correctness.

2. TOOL SUPPORT : IBM RATIONAL RHAPSODY

This paper uses a tool called IBM Rational Rhapsody Developer for Java 7.6.1 to build and execute the state machines [3]. The generic design patterns are customized and validated using the tool. The functionality of the components in the design patterns can be depicted in Rational Rhapsody by using the Rhapsody’s Action language which is similar to java and Event handling infrastructure.

IBM Rational Rhapsody’s action language can be used to capture actions and to execute the model. This action language can be used to make the diagrams executable by representing the actions each object performs and also the messages the objects pass to other objects when an event occurs. The message passing can be represented in state charts, by depicting the message passing and the respective transitions. The action language is similar to Java, except there are a few additional reserved words. 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() method that is called to implement the appropriate task. Code for these methods can be written using the action language. 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 within that state are performed. Thus, executable state charts represent the functional behavior of the components of the system. However the coding required for IBM Rational Rhapsody differs by the Object eXtended Framework (OXF) involved which depends on the Developer Edition (Java or C++). That is the action language semantics and syntax differs based on the OXF.

IBM Rational Rhapsody is an excellent tool to create dynamic UML diagrams using Real-time UML that is UML 2.0. These executable state charts and Object Model Diagrams can be validated using Rational Rhapsody. Rhapsody is also used to generate code for the diagrams.

3. UML 2.0 AS ARCHITECTURAL DESCRIPTION LANGUAGE (ADL)

The Unified Modeling Language (UML 1.0) [4] was first introduced as a formal graphical language to represent the system as static diagrams but was later revised to represent the functionality of components of real-time systems as UML 2.0 also called as Real-Time UML.

Architecture Description Language (ADL) is defined as “a language (graphical, textual, or both) for describing a software system in terms of its architectural elements and the relationship among them” [4]. UML is widely accepted language by practitioners. This paper uses UML 2.0 to represent the components of the systems in terms of an Object Model diagram and state chart diagrams. The UML 2.0

diagrams are represented using the Component and Connector views (C&C views, for short) [5]. They present 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 UML extensibility mechanisms (i.e., stereotypes, tagged values, constraints) are used to interpret the functionality of the system in the diagrams [6].

In UML 2.0, the components are created as Composite classes 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).

4. SOLAR TERRESTRIAL RELATIONS OBSERVATORY (STEREO)

The STEREO mission is a two year mission from NASA with a goal to provide the first ever three dimensional images of the Sun by studying the nature of Coronal Mass Ejections (CME). The mission involves using two nearly identical three-axis stabilized spacecraft in heliocentric orbit, which is an orbit around the sun. Since the spacecraft is far away from Earth, the STEREO FSW relies less on real-time ground commanding and more on autonomous functionality. Additionally, since STEREO operates in a heliocentric orbit it requires guidance and control algorithms along with propulsion hardware to achieve and maintain its orbit.

The STEREO spacecraft contains four payload instrument packages to accomplish its scientific mission. The payload packages are In-situ Measurements of Particles And CME Transients (IMPACT), PLAsma and SupraThermal Ion Composition (PLASTIC), STEREO/WAVES (S/WAVES), and Sun Earth Connection Coronal and Heliospheric Investigation (SECCHI). The IMPACT package measures solar wind electrons, energetic electrons, protons, heavier ions, and the in situ magnetic field strength and direction. PLASTIC measures the composition of heavy ions in the ambient plasma, protons, and alpha particles. S/WAVES measures the generation and evolution of traveling radio disturbances. Finally, SECCHI uses remote sensing imagers and coronagraphs to track CMEs.

The payload instruments collect data 24 hours a day, even when the spacecraft is in communication with the ground. During events of interest, the FSW must also enable the appropriate instruments to collect data at higher sampling rates. The FSW is responsible for performing some processing on the data such as data compression and formatting the data into telemetry packets. Additionally, the FSW must collect and store data from all the payload packages. The data is pushed from the instrument data buffers to the FSW during predetermined time intervals and using predefined data rates.

STEREO maintains its orientation in space using a three-axis stabilization technique, as opposed to a spin stabilization technique. In STEREO's three-axis stabilization technique, reaction wheel assemblies (RWAs) are mounted on various sides of the spacecraft and the appropriate RWAs are fired to make slight changes to the spacecraft's orientation. To adjust the spacecraft's attitude, adjustments are managed by firing thrusters to push STEREO to the proper attitude. Attitude determination and control is managed autonomously onboard the spacecraft by the FSW. The FSW determines the attitude and orientation using measurements from one star tracker, six sun sensors, and one Inertial Reference Unit (IRU). If the FSW determines that the attitude and orientation are out of the acceptable range, then it must determine and send the appropriate commands to the RWAs and/or thrusters to adjust the spacecraft's attitude.

STEREO uses two movable solar array appendages to generate power. The FSW is responsible for positioning the solar arrays toward the sun. Onboard power is controlled using a Power Distribution Unit (PDU). To maintain a consistent temperature, STEREO spacecraft uses both active and passive means of thermal control. The active measurers include thermistors and electric heaters to ensure the spacecraft remains a consistent temperature throughout the spacecraft since one side of the STEREO is facing the Sun and the other does not. The FSW is responsible for monitoring the spacecraft temperatures and sending commands to the appropriate heaters to adjust the temperature. STEREO downlinks its data once per day to the ground through NASA's Deep Space Network (DSN) station in Canberra, Australia. To communicate with the ground, STEREO contains a Low Gain Antenna (LGA), Medium Gain Antenna (MGA), and High Gain Antenna (HGA) that will be used depending on where the spacecraft is in orbit. The FSW is responsible to selecting and using the right antenna at the right time. The LGA and MGA are fixed, however the STEREO FSW is responsible for autonomously controlling the HGA so that is pointed at Earth.

Based on the functionality of STEREO ten design patterns have been identified to depict the functionality of STEREO. The design patterns identified are listed in the table 1.

Table 1. STEREO Design Patterns

Feature	Design Pattern
High Volume Command Execution	**STEREO Hierarchical Control Design Pattern**
High Volume Telemetry Storage and Retrieval	**STEREO Compound Commit Design Pattern**
High Volume Telemetry Formation	**STEREO Pipes and Filters Design Pattern**
Quick Check	**STEREO Sanity Check**
Event Driven Payload Data Collection	**STEREO Payload data Client Server Design Pattern**
Ground Driven Housekeeping Data Collection	**STEREO Housekeeping data Multiple Client Multiple Server Design Pattern**
Event Driven Housekeeping Data Collection	**STEREO Housekeeping data Client Server Design Pattern**
Housekeeping Data Checks	**STEREO Housekeeping Checks Multicast**
Spacecraft Clock	**STEREO Spacecraft Clock Multicast**
Memory storage Device fault Detection	**STEREO Memory Storage Device Watchdog.**

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

4.1. Hierarchical Control Design Pattern:

STEREO FSW must interact with ten components to implement its functionality. Hierarchical controller would be appropriate to implement such functionality where a separate controller is identified for components implementing similar actions. Example a Payload subsystem controller is identified to control the behavior of the four payload devices of STEREO. Thus a hierarchical control design pattern best suites the working of STEREO.

4.2. Compound Commit:

The data collected from different instruments of STEREO is to be stored or retrieved in a "all-or-nothing" methodology. That is either all the telemetry data is to be stored or nothing is to be stored by the components, similarly for retrieval of telemetry data.

4.3. 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.

4.4 Sanity Check

The Sanity Check design pattern is a pattern to improve reliability and ensure that the system performs more or less as expected. If a problem is detected, then the system is put into a failsafe state, which is a state that is always known to be safe. This design pattern is included because it provides a level of reliability to the Pipes and Filter design pattern. This design pattern is suitable to use on DRE applications that have reliability requirements but do not have high availability requirements.

4.5 Payload Data Multiple Client Multiple Server Pattern

The STEREO spacecraft contains four payload instrument packages to accomplish its scientific mission. The payload packages are In-situ Measurements of Particles And CME Transients (IMPACT), PLAsma and SupraThermal Ion Composition (PLASTIC), STEREO/WAVES (S/WAVES), and Sun Earth Connection Coronal and Heliospheric Investigation (SECCHI). The IMPACT package measures solar wind electrons, energetic electrons, protons, heavier ions, and the in situ magnetic field strength and direction. PLASTIC measures the composition of heavy ions in the ambient plasma, protons, and alpha particles. S/WAVES measures the generation and evolution of traveling radio disturbances. Finally, SECCHI uses remote sensing imagers and coronagraphs to track CMEs. A separate client and server for each of the payload instruments are created to collect the information whenever the controller signals to collect.

4.6. 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 ten components of STEREO.

4.7. Housekeeping Data Client-Server Design Pattern

This design pattern is used to represent the collection of housekeeping data from the client when an event occurs. When a request for a particular housekeeping data occurs in the form of an external event, this design pattern collects the information.

4.8. Housekeeping Data Checks Multicast

The housekeeping data checks design pattern performs certain checks on the housekeeping data at regular intervals and multicasts the messages to various components.

4.9. Spacecraft Clock Multicast Design Pattern

This pattern is used to send time signals to the Controller and input and output components of the system.

4.10. Memory Storage Device Watchdog Design Pattern

The memory storage device in STEREO 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.

These ten design patterns are implemented in IBM Rational Rhapsody.

5. IMPLEMENTATION

Two design patterns are explained based on the functionality of STEREO. The UML diagrams built and validated using IBM Rational Rhapsody.

5.1 STEREO Hierarchical Control Design Pattern

The Hierarchical Control Design Pattern is selected as the components in STEREO cannot be controlled by a Centralized Controller. Also STEREO being away from the Earth, most of the processing needs to be done in the satellite itself. So different subsystems have been identified and each

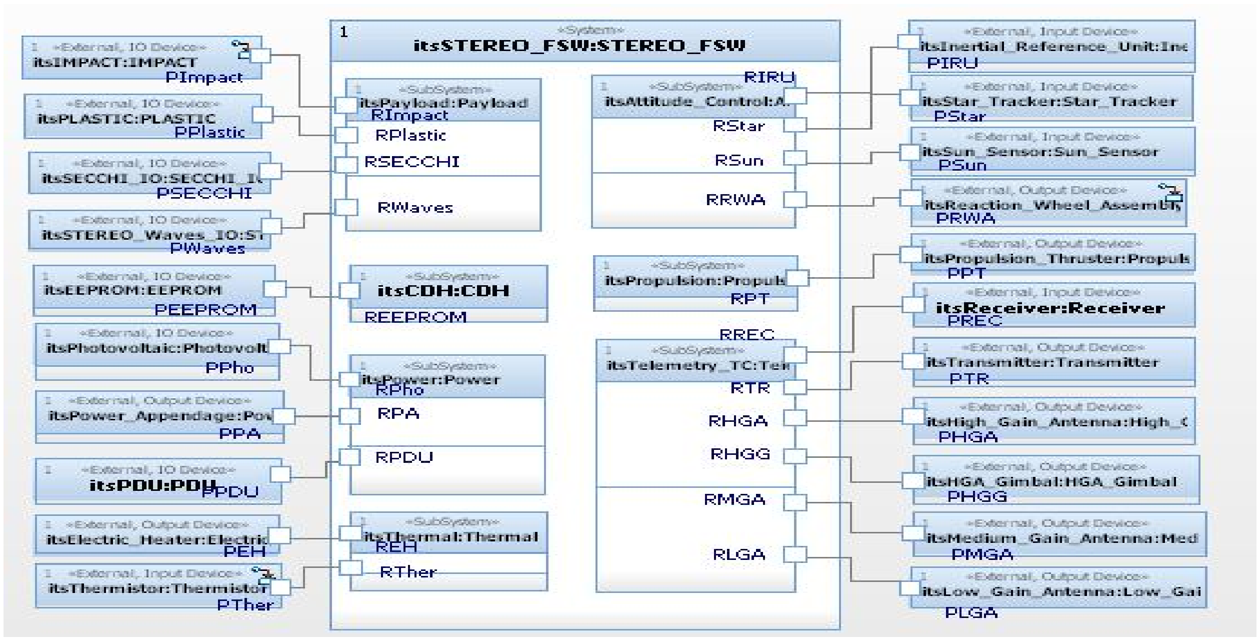
«System» itsSTEREO_FSW:STEREO_FSW
itsIMPACT:IMPACT PImpact
itsPLASTIC:PLASTIC PPlastic
itsSECCHI_IO:SECCHI_IO PSECCHI
itsSTEREO_Waves_IO PWaves
itsEEPROM:EEPROM PEEPROM
itsPhotovoltaic:Photovolt PPho
itsPower_Appendage PPA
itsPDU:PDU PPDU
itsElectric_Heater:Electric PEH
itsThermistor:Thermistor PTher
«SubSystem» itsPayload:Payload RImpact RPlastic RSECCHI RWaves
«SubSystem» itsCDH:CDH REEPROM
«SubSystem» itsPower:Power RPho RPA RPDU
«SubSystem» itsThermal:Thermal REH RTher
«SubSystem» itsAttitude_Control RIRU RStar RSun RRWA
«SubSystem» itsPropulsion:Propulsion RPT
«SubSystem» itsTelemetry_TC RREC RTR RHGA RHGG RMGA RLGA
«External, Input Device» itsInertial_Reference_Unit PIRU
«External, Input Device» itsStar_Tracker:Star_Tracker PStar
«External, Input Device» itsSun_Sensor:Sun_Sensor PSun
«External, Output Device» itsReaction_Wheel_Assembly PRWA
«External, Output Device» itsPropulsion_Thruster:Propulsion PPT
«External, Input Device» itsReceiver:Receiver PREC
«External, Output Device» itsTransmitter:Transmitter PTR
«External, Output Device» itsHigh_Gain_Antenna:High_Gain PHGA
«External, Output Device» itsHGA_Gimbal:HGA_Gimbal PHGG
«External, Output Device» itsMedium_Gain_Antenna:Med PMGA
«External, Output Device» itsLow_Gain_Antenna:Low_Gai PLGA

Figure 1 Object Model Diagram for Hierarchical Controller

of the subsystem takes care of the working of the components under it. This is depicted in the Object Model Diagram of the Hierarchical Control Design pattern Figure 1.

Each of the components is interconnected to their respective subsystems by the use of ports. IBM Rational Rhapsody is an excellent tool to represent the components of the system and their interconnections using the Component and Connector view of UML 2.0. The ports act as the interface to enable the component to interact with the external world. Each of the ports realizes interfaces which can be Provided interface or Required interface, which are not shown in the diagram for readability. The interfaces should be specified as part of the contract feature of the port.

The stereotypes may be used to identify the components as Input, Output or IO Component. Next a state chart diagram is built for each of the components as shown below figure 2, figure 3 and figure 4.

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

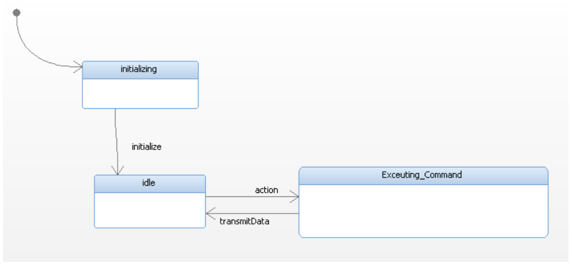

Figure 2 State chart diagram for Output component

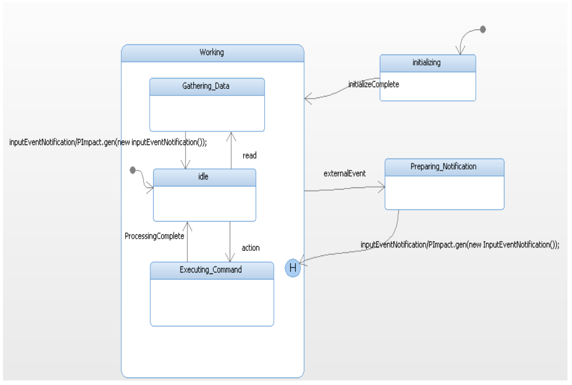

Figure 3 State chart diagram for IO Component

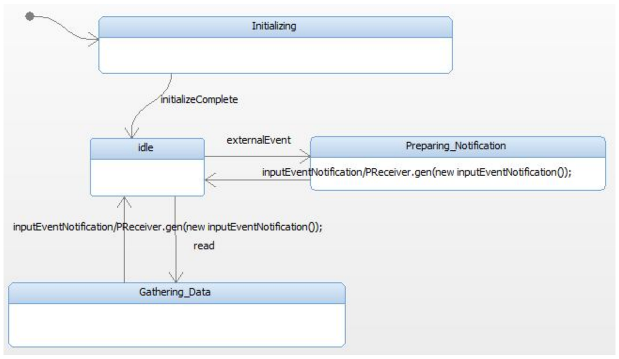
Initializing
initializeComplete
idle
externalEvent
Preparing_Notification
inputEventNotification/PReceiver.gen(new inputEventNotification());
inputEventNotification/PReceiver.gen(new inputEventNotification());
read
Gathering_Data

Figure 4 State Machine for Star_Tracker_IC

5.2 Memory Storage Watchdog Design Pattern

The Watchdog design pattern (Douglass 2003) is a lightweight design pattern to improve system reliability by making sure the processing is going as expected. This design pattern is included because it provides a lightweight approach to providing reliability. This pattern ensures the reliable working of the memory storage device that is EEPROM.

When the process is going as expected, the Watchdog receives stroking messages from the component it is monitoring. If it does not receive a stroking message within a given amount of time, the watchdog assumes a fault has occurred and sends out an alarm.

The FSW Memory Storage Device Watchdog Executable Design Pattern contains the components necessary to monitor the memory storage device for faults. The components and their behavior are common at the FSW and thus there are no SNOE specific customizations to this design pattern.

A *watchdog,* used in common computing parlance, is a component that watches out over processing of another component. In SNOE Watchdog is used to check the memory storage device for faults. Its job is to make sure that nothing is *obviously* wrong, just as a real watchdog protects the entrance to the henhouse without bothering to check if in fact the chickens inside are plotting nefarious deeds. The watchdog ensures that the memory storage device works properly. It checks the device and its functionality in predefined intervals.

The simplicity of the Watchdog Pattern is apparent from Figure 5. The *Actuator Channel* operates pretty much independently of the watchdog, sending a *liveness* message everyso often to the watchdog. This is called *stroking* the watchdog. The watchdog uses the timeliness of the stroking to determine whether a fault has occurred.

- ***Actuation Channel***

This is the channel that contains components that perform the end-to-end actuation required by the system. "End-to-end" means that it includes the sensing of control signals from environmental sensors, sequential or parallel data processing, and output actuation signals. It contains no components in common with the *Watchdog.*

- ***Actuation Data Source***

The *Actuation Data Source* is the source of sensed data used for control of actuation.

- ***Actuator***

The *Actuator* actor is the actual device performing the actuation.

- ***Data Transformation***

As in the other patterns, these components process the sensing data in a sequential fashion to compute the ultimate actuation output. This can be done with a single datum running all the way through the channel before another is acquired or with multiple data in various stages of processing simultaneously to provide a serial or parallel *Actuation Channel,* respectively.

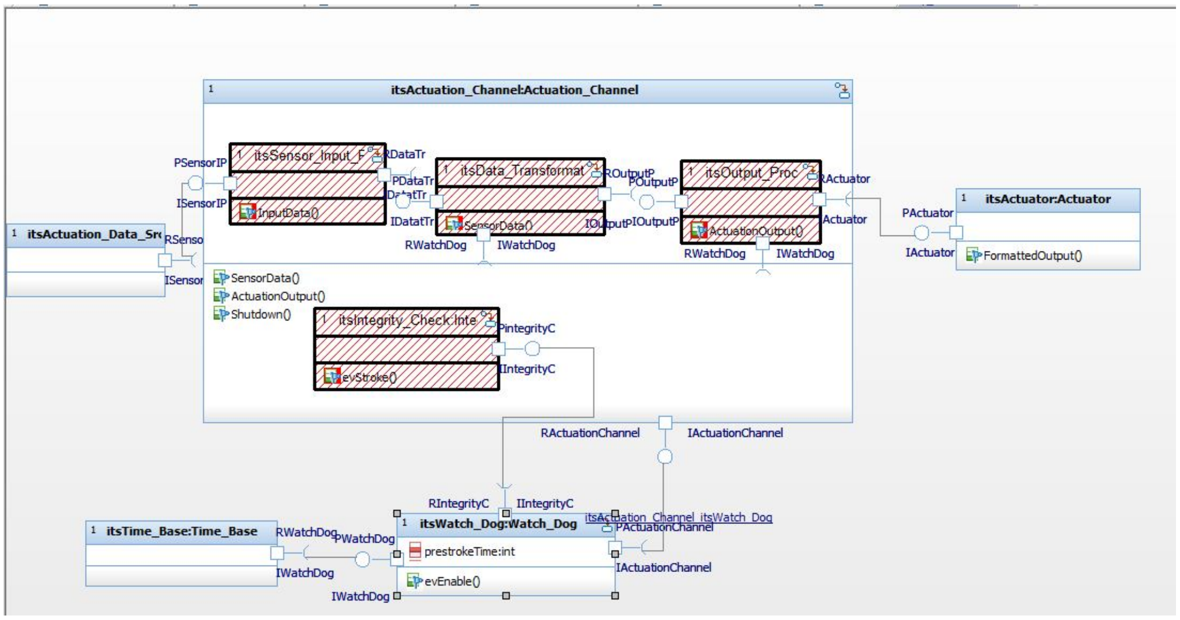

Figure 5 Object Model Diagram of Watchdog Design Pattern

- ***Sensor Input Processing***

The *Sensor Input Processing* component is a device driver for the *Actuation Data Source* actor. It performs any initial formatting or transformations necessary for the particular *Actuation Data Source* sensor.

- ***Integrity Checks***

This component is (optionally) invoked on every valid stroke of the *Watchdog*. This can be used to run a periodic Built In Test (BIT), check for stack overflow of the tasks, and so on**.**

- ***Output Processing***

This is a device driver for the *Actuator* actor. It performs any final formatting for transformations necessary for the particular *Actuator*.

- ***Timebase***

The *Timebase* is an independent timing source (such as an electronic circuit) used to drive the *Watchdog*.

- ***Watchdog***

The *Watchdog* waits for a stroke event sent to it by the components of the *Actuation Channel*. If the stroke does occur within the appropriate timeframe, the *Watchdog* may command integrity checks to be performed. If it does not, then it shuts down the *Actuation Channel*.

Some watchdogs check that the stroke comes neither too quickly nor too slowly. The statechart for such a time-range watchdog is shown in Figure 6. For some systems, protection against a timebase fault is safety-critical. In such cases, it is preferable to have an independent timebase. This is normally a timing circuit separate and independent from the one used to drive the CPU executing the *Actuation Channel*.

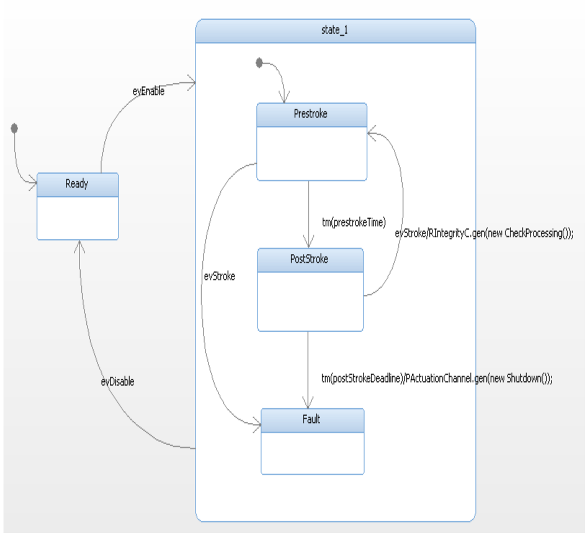
state_1
evEnable
Ready
Prestroke
tm(prestrokeTime)
evStroke/RIntegrityC.gen(new CheckProcessing());
PostStroke
evStroke
evDisable
tm(postStrokeDeadline)/PActuationChannel.gen(new Shutdown());
Fault

Figure 6 State machine for Watchdog

As mentioned before, if the watchdog is to provide protection from timebase faults, a separate electronic circuit *must* supply an independent measure of the flow of time. This means an independent timing circuit, usually driven by a crystal, but the timebase may be driven by an R-C circuit. Note, however, that the watchdog detects a *mismatch* between the two timing sources.

When the watchdog is stroked, it is common to invoke a BIT (Built In Test) of some kind to ensure the proper execution of other aspects of the system. These actions can either return a Boolean value indicating their success or failure, or may directly cause the system to shut down in the case of their failure. For example, the watchdog may execute an action on the evStroke transition (see Figure 6) that checks for stack overflow and performs CRC checks on the executing application software. If it does a similar check on the application data, it must lock the data resources during this computation, which can adversely affect performance.

When watchdog fires because it hasn't been stroked within the specified timeframe, it invokes some safety measure, normally either shutting down the system or causing the system to reset.

Similarly, the *Object Model Diagrams* and *state machines* for all the identified design patterns are developed and validated.

6. RESULTS

This paper validates the design patterns using the tool IBM Rational Rhapsody. Rational Rhapsody generates the code for the design patterns and validates the design patterns using 'build' option. Thus the functionality of design patterns can be verified during the design phase and thus reduce the number of anomalies in flight software. This validation of design patterns for functional correctness was not possible in static UML diagrams using UML 1.0. Rational Rhapsody also enables the animation of statecharts by generating events to check the behavior of the component. The figure 7 is an example of animated statechart of client component where the bright colored state indicates the present state of the component after the requestNeeded event is generated. Events can be generated manually while executing the state charts and thus transitions between the states can be checked. There by ensuring about the functionality of the components.

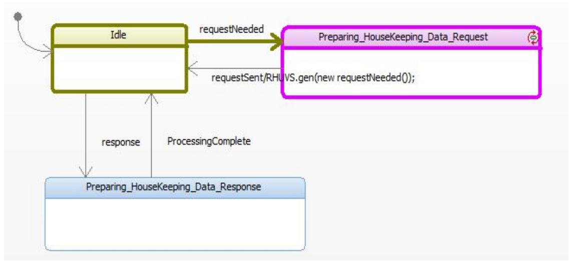
Idle
requestNeeded
Preparing_HouseKeeping_Data_Request
requestSent/RHHVS.gen(new requestNeeded());
response
ProcessingComplete
Preparing_HouseKeeping_Data_Response

Figure 7. Animated State Chart for Client

Thus the functionality of every component in the design pattern can be validated to build an error free Architecture.

7. CONCLUSIONS

This paper uses generic architectural design patterns in the DRE domain to build the architecture of a satellite. The design patterns are customized to suit the functionality of the satellite and are also validated to reduce the risk of errors occurring after implementation. Also the design patterns are made executable to be used in the future for any other DRE domain.

8. FUTURE ENHANCEMENTS

This paper can be extended by including the performance validation. Also various other options of Rhapsody can be used to better represent the functionality and performance of design patterns.

9. REFERENCES

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

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

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

[4] B.Bharathi, Dr.D.Sridharan. 2009. *UML as an Architecture Description Language,* International Journal of Recent Trends in Engineering.

[5] Software Architecture Description & UML Workshop, Hosted at the 7th International Conference on UML Modeling Languages and Applications <<UML>> 2004, October 11-15, 2004, Lisbon, Portugal.

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