ABSTRACT

Robot autonomy is a capability required to support current and future space robotics missions. Planetary exploration robots and unmanned reusable space vehicles will require a high level of autonomy to perform tasks more efficiently. The Canadian Space Agency has designed, implemented and tested different autonomy techniques on typical autonomous robotics scenarios. Over the time, CSA have analyzed the strong and weak points of Finite State Machine (FSM), Hierarchical Task Networks (HTN) and Goal Decomposition Hierarchy (GDH) techniques. Based on experiments, we merge the strong points into a new user-friendly oriented toolbox. The ARGO Cortex Toolbox allows the implementation of hierarchies of finite state machines. By using parameters, state machines are designed modularly in order to be reused in different contexts. The Cortex development environment provides features to design, manage, generate code, execute, deploy and monitor execution of autonomy engines.

INTRODUCTION

The recent re-assessment of the priorities of several space organizations around the world has resulted in a spectacular increase in the number of robotic missions being planned. In the area of on-orbit servicing, missions such as Orbital Express, TECSAS and the Hubble Repair mission are either being executed or planned on a very tight schedule. NASA’s new exploration initiative is calling for robotic missions to the Moon and Mars at a very frequent rate over a long period. One of the key technologies that will be required to succeed towards the ambitious objectives that are being set internationally will be to streamline the operations of future space missions.

Currently, space robotic operations are very human-intensive throughout the planning and verification phase. Armies of ground staff are required to plan-out and rehearse operations in advance trying to assess the impact of all possible outcomes and anomalies. In the case of orbital robotics, this level of operator involvement is maintained throughout the execution phase as several operators/experts monitor the mission in real-time. In all cases, the highest level of autonomy/automation attained is the remote execution of a very linear, extensively rehearsed sequence of commands. Operations concepts such as that of the International Space Station, and even that of the Mars Exploration Rovers, result in expensive operations that we will not be able to afford if the plans being put forward are to be realized.

Over the last decade, many approaches have been implemented to circumvent these problems and allow controlling space-based robots from ground stations[1][2][7][9]. In most cases, the focus was on developing architectures to enable ground control of space-based robots rather than aiming specifically at enhancing efficiency of operations. Recently, research was conducted to develop software architectures to incorporate some elements of on-board autonomy [3][4][5][6][10].

This paper describes recent advances made by the Canadian Space Agency (CSA) in the development of software tools for ground control and autonomous robotics. The Autonomous Robotics and Ground Operations (ARGO) software suite is briefly presented. Special attention is dedicated to the description of the Cortex on-board autonomy toolbox. The basic principles, concept of use and software architecture of the Cortex toolbox are described in detail.
THE ARGO FRAMEWORK

The ARGO project has two main objectives:

• Augmenting the on-board capabilities of space-borne systems in terms of autonomy and
• Reducing of the costs associated with space operations by seamlessly integrating the space operations process from planning through verification, execution up to post-flight analysis.

The target applications of the ARGO framework [8] cover the full spectrum of autonomy from supervisory control such as might be expected for ISS robotics to more autonomous operations such as would be encountered in planetary exploration missions. It also covers the full range of space robotic applications from orbital manipulators to planetary exploration rovers. One of the important features of the ARGO framework is that it does not provide a universal architecture for ground control and autonomy. Instead, ARGO provides a set of toolboxes that can be assembled in a variety of manners depending on the application and its requirements. To facilitate the re-use of software, the design is modular and portable to the maximum extent possible.

Several toolboxes already exist within the ARGO framework. The Resource Toolbox is used to distribute capabilities among different platforms. It is the central mechanism used for remotely accessing devices and connecting to telemetry streams. The Log Toolbox provides a standardized mechanism for the dissemination of discrete log messages between applications. These messages are used to monitor the health and status of all systems.

A Re-configurable Ground Control Station has been implemented, which allows operators to easily build an operator interface to send commands and monitor telemetry. This interface is intimately tied to the Resource and Log Toolboxes. Using this tool, an operator can generate a control station in a few minutes and even customize the control station on the fly.

Finally, the Cortex Toolbox provides a set of tools to implement on-board autonomy software based on the concept of hierarchical finite state machines. The Cortex Toolbox allows an operator to graphically generate the behaviours to be implemented on the remote system. It automatically generates the code to be uploaded and it can be used to debug and monitor the execution of the autonomy software on-line and off-line.

AUTONOMY TOOLBOX: CORTEX

Modular Hierarchical Finite State Machines

The Cortex toolbox uses Finite State Machines (FSM) to implement reactive autonomy behaviours. FSMs are used to represent a system using a finite number of configurations, called states, defined by the system parameters or its current actions. Fig. 1 depicts a simple FSM. The states are Starting_Motion, Turning, and Stopping_Motion. In Cortex, actions to be taken on a state entry, re-entry and exit can be defined. For example, when entering the Turning state, the current robot azimuth angle could be recorded to serve as a starting point for the destination angle computation.

The system can transition from one state to another based on its current state, conditions and outside events. In Cortex, conditions on transitions are called Guard, and are implemented as statements that can be evaluated as being either true or false. Outside events are called Triggers. Such events trigger the evaluation of the FSM Guards and enable transition to occur. In Fig. 1 for example, the system will transition from Starting_Motion to Turning once robot motion is confirmed, or from Turning to Stopping_Motion when the compass is functioning correctly and its readings confirms the robot has turned by the specified angle. In this particular FSM, no specific Triggers have been defined, so any Trigger will make the FSM evaluate its Guard.

A set of states connected together by transitions forms a machine. In Fig. 1, the Turn_on_Spot_by_Angle block is an FSM that implements a behaviour that has a mobile robot turn on the spot. The FSM also contains parameters it uses to take decisions (such as the current robot heading and the commanded angle of the turn) or on which it acts (the robot itself).

In Fig 1., Starting_Motion is represented as a single state. However the logic to be used in this state could be complex. In order to represent the state logic, we could decide to use a sub-FSM in its place. The result is a Hierarchical Finite State Machine (HFSM) as shown in Fig. 2.
In this case, we can make the **Starting Motion** sub-FSM modular by specifying as an input parameter, the ID of the robot on which to apply its logic. This allows this sub-FSM to be reused in another higher level FSM. This is the strategy used by Cortex to provide FSM modularity and reusability.

**Concept of Use**

Cortex is not only an autonomy database builder, it is a complete framework that allows the developers to design, create, execute, deploy and monitor autonomous engines. Instead of using different applications and tools, Cortex provides a single program to manage these different steps (Fig. 3). Depending on the situation, the user may develop low level FSMs that interact with the hardware drivers or he may develop higher FSMs to implement autonomous behaviours.

The first step consists in creating a new project. A Cortex project includes a root FSM that represents the highest level of refinement. Once created, the developer may insert states, junctions and transitions. The developer may specify parameters to define inputs, outputs and local variables. In complex logical cases, the developer may hide details by using sub FSMs. These sub FSMs correspond to lower level of refinement. They contain their own parameters, states, transitions, junctions and sub FSMs. To indicate the transition conditions, the developer edits the **Guard** by defining logical expressions. These expressions will be evaluated when triggers are submitted and received during the execution. The triggers are defined globally to the project and are associated to the transitions.
Cortex uses eXtensible Markup Language (XML) format as file storage standard to store Cortex projects. It contains the specifications and the graphical representation of the FSMs included in the project. The graphical representation is stored in Scalable Vector Graphics (SVG) format and allows the developer to view the Cortex projects using any Internet Explorer equipped with a SVG plug in.

To increase the complexity of the autonomy, the developer may use copy/cut/paste capability. The developer may also make references to external Cortex projects to reuse existing functionalities. This is useful when dealing with complex autonomy problems. This approach allows the development of Cortex Libraries and permits a divide-and-conquer approach to solve problems. These libraries may then be reused in different contexts.

Once the FSMs have been developed, the developer invokes the Cortex Coder to generate the real-time code. This code is the conditional logic equivalent of the Cortex FSM specifications and is independent of Cortex environment.

The code is then compiled and errors are reported through an error module. Cortex highlights the FSM components responsible of the errors. The real-time code may be used outside of the Cortex framework.

The developer may then start a local executor and tests the behaviour of the FSM. By using the executor panels, the developer can submit triggers to the FSM to tests the several states. FSM parameters may be monitored and the operator may change their value at run-time. This allows verifying and validating Cortex projects. FSM graphical representations are animated in real-time to provide a monitoring mechanism to the developer. Last transitions and current states are highlighted. The developer can then deploy the Cortex code onto the embedded platform. A Remote Executor allows the developer to transfer the code and to control it remotely. The same interface is used to submit triggers and to receive state and parameters changes. All this is done transparently through the ARGO Resource Toolbox [8].

All state and parameter changes are logged either locally or remotely. The log records all the execution history of the FSMs, which allow the user to playback the execution for analysis purposes. The developer may use the ARGO Log Viewer to visualize the sequence of events on a time scale.

Finally, the Cortex report generator is used to automatically generate HTML reports on all the FSM components included in a project. This documentation has all the information required by developers to understand and integrate the FSMs into other applications.

**Cortex Features**

Cortex's main goal is to provide a framework to develop efficiently complex autonomy engines while minimizing work on interfacing and low-level programming. From the start user-friendliness was identified as one of the main Cortex driver. The developer should not spend a lot of time on interfacing. He should rather work on the development of the autonomy engine itself.
Integrated Environment

Cortex modules are all bound into a single Graphical User Interface (GUI). This interface provides all the features required to execute all the steps mentioned in the previous section: It provides panels to

- edit and create Cortex projects;
- generate and compile real-time code;
- deploy the real-time code onto the target system;
- execute, command, control and monitor local and remote instances of the real-time code;
- playback previous state and parameter changes;

Variable Level of Autonomy

As mentioned previously, the main driver of Cortex is being able to manage complex autonomy projects. To create levels of refinement, the developers use sub FSMs. Parameters are used to specify the inputs and outputs of the FSMs. Using copy-paste capability, the developer may create multiple instances of the same FSM. Furthermore, the developer can modularize Cortex projects. The root FSM parameters and project triggers define the inputs and the outputs of a project. This modularity allows developers to create Cortex project libraries to be invoked by other Cortex FSM.

Multi-Project Sessions

All this can be done simultaneously in a multi-project environment. The developer may edit, create, control and monitor more than one project at the same time. He can copy from a project to another, prototype a FSM in a project panel, test it and at last upgrade another project with the newly tested FSM.

Distributed Autonomy

Cortex supports the distribution of autonomy among multiple systems: for example across several robots or throughout a distributed ground control station. A state from a Cortex project may submit a trigger to another Cortex project using the ARGO Resource Toolbox. It is also possible for multiple operators to control, command and monitor the same autonomy engine.

System Targeting

Cortex Coder generates real-time code mainly composed of conditional logic statements. It is free of threads and performs computation only on trigger invocations. It is self-contained and does not require the Cortex framework to be executed. The developer may decide to take the real-time code and integrate it by hand in his application. The Cortex Coder currently supports Java but its architecture will support C++, VHDL and Ada languages.

Cortex Architecture

The architecture provides modules that implement the functionalities of the Cortex framework. Fig. 4 shows the two parts of the Cortex architecture. The Development Environment refers to the environment used by the developers to create, design, generate, deploy, control, command and monitor autonomy engines. The Target System Environment refers to the system where the autonomy engine is running. The modules are portable and have been developed with Java. They have been tested on Windows, Linux and Solaris. In addition, ARGO provides Java packages to interface with non-java code (e.g. C, C++).

![Fig. 4: Cortex Architecture](image-url)
Development Environment

The main module of the Development Environment is the Graphical User Interface (GUI). The GUI provides all the panels required to create, generate, deploy, command and monitor Cortex projects. It is the core of the Cortex framework and provides drag-and-drop feature to build FSM and supports copy/paste operations. It makes the interaction between the developer and the Cortex Database API transparent. This API defines the Cortex project specifications such as the name of the FSM components, their connectivity's and execution parameter settings.

Coder

The Cortex Coder task consists in generating the real-time code according to the Cortex Database specifications. The code generated is composed of simple overlapped if-then-else statements. Methods are generated for each project trigger. These methods are called whenever a trigger is received by the FSM. Methods are also generated for each input and output parameters of the project (root FSM parameters) to allow setting and getting the parameter value at runtime. The Cortex Coder also generates HTML reports.

Executor

The Cortex Executor provides methods to submit triggers and to set and get parameter values. It implements an event-listener architecture to notify state changes and parameter changes. These events are reported to the operator through the GUI. Using the ARGO Resource Toolbox, the Cortex Executor may be instantiated remotely. Multiple GUI may establish a connection, interact and monitor the same remote Cortex Executor.

APPLICATIONS

Planetary Exploration

The first use of the Cortex Toolbox in an operational scenario has been demonstrated in the Mobile Robotic Testbed (MRT) at the Canadian Space Agency. In this scenario, a rover is placed on a terrain emulating the surface of Mars, and is asked by an operator to go to a specified location. The rover is a skid-steered P2AT platform from ActiveMedia and is equipped with sonar arrays and optical sensors for obstacle detections, an Inertial Measurement Unit and magnetic compass for azimuth determination and a scanning LIDAR (Iliris-3D from Optech).

A HFSM built using Cortex, initialise the robot sub-system and then ask the LIDAR to take a scan of the terrain to be traversed. This scan is used to localise the rover in the environment. Once the location of the rover is known, Cortex invokes a path planner using the current location and destination together with the triangulated map produced using the LIDAR data. The path planner returns a trajectory that is safe for the robot to get to the destination. Cortex starts a guidance system that actively controls the robot along the trajectory. While the trajectory is being followed, the HFSM monitors sensors for obstacle and stops the robot if required. In such a case, Cortex ask the LIDAR to take another scan and the trajectory "plan and follow" cycle is repeated until the destination is reached, or the destination is found to be unreachable.

Space Robotics Servicing

Another scenario driving the requirements of Cortex is autonomous on-orbit servicing of failed/failing spacecraft. In this case, Cortex is used to manage the performance of the short-range rendezvous and capture. The CSA Automation Robotics Testbed (CART) is used to demonstrate such functionalities. This testbed is composed of two 7 degrees-of-freedom arms. In the servicing scenario, one arm emulates the motion of the spacecraft to be serviced and the other emulates the motion of the servicer. The capture operation is a high-risk task and the following sequence is implemented in a Cortex engine.

1. Enter into short range rendezvous mode after completion of the long-range rendezvous;
2. Maneuver toward the serviced satellite and align with the grasping interface;
3. Upon completing alignment with the grasping interface, turn the robotic arm into force control mode;
4. Wait for the serviced spacecraft’s Attitude Orbit Control System to be turned off;
5. Perform the capture by grasping the satellite through the grasping interface.

Transitions between phases of the operation are triggered by sensory events. The Cortex engine considers anomalies such as Blinding and loss of the communication. While executing, the operator may interact with the engine by submitting triggers or changing parameter values.
CONCLUSION

To address the need of future space robotics missions, the ARGO framework aims at generating software toolboxes to enable ground control of space robot with varying levels of autonomy. It also aims at reducing the costs associated with space operations by seamlessly integrating the space operations process from planning through verification, execution up to post-flight analysis. Toolboxes can be configured to handle space robotic applications ranging from orbital manipulators to planetary rovers.

This paper describes the ARGO project and particularly the Autonomy Toolbox: Cortex. Cortex's goal is to provide a user-friendly integrated environment to create, validate, deploy and monitor autonomy engines. Two applications of Cortex (Mars Robotic Testbed and Space Robotics Servicing) are described.

REFERENCES