Implementation of Online Control Customization within the Open Control Platform

Raktim Bhattacharya * Gary J. Balas †

Department of Aerospace Engineering & Mechanics,

University of Minnesota,

Minneapolis, MN-55455

Abstract

Integration of MATLAB with the Open Controls Platform for three F-16 flight control examples is presented. The flight control examples implement LQR control, LPV regulation and waypoint tracking of a F-16 airplane model, respectively. The highlights of this paper are; Constructing MATLAB based flight control examples in the OCP framework; Ground work for a new code generation based tool that will translate SIMULINK blocks to OCP components; Discussion on automatically generating asynchronous systems from dataflow specifications obtained from SIMULINK model files.

1 Introduction

Today, the role of a control algorithm is evolving from static designs, synthesised off-line to dynamic algorithms that adapt in real-time to changes in the controlled system and its environment. The paradigm for control system design and implementation is also shifting from a centralised, single processor framework, to a decentralised, distributed processor implementation framework. Distribution and decentralisation of services and components is driven by the falling cost of hardware, increasing

*Graduate Assistant, raktim@aem.umn.edu

†Professor, balas@aem.umn.edu

computational power, increasingly complex control algorithms and development of new, low cost micro sensors and actuators. A distributed, modular hardware architecture offers the potential benefit of being highly reconfigurable, fault tolerant and inexpensive. Modularity can also accelerate development time of products, since groups can work in parallel on individual system components. These benefits come with a price, the need for sophisticated, reliable software to manage the distributed collection of components and tasks.

Communication within a distributed, decentralised environment becomes a significant issue. Hardware components and software processes operate in synchronous and asynchronous modes. These processes have to communicate with one another with a well defined protocol to effectively control the system. Software tools needed for a distributed, realtime control architecture include: real-time execution, adaptive task scheduling, task synchronisation, communication protocols and adaptive resource management. Hence software and its interaction with the controlled system will play a significantly larger role in the control of real-time systems. This drive towards distributed, dynamic control systems requires establishment of tighter ties between the controls and computer science community for these systems to be successful. In 1999, DARPA initiated the Software Enabled Control (SEC) program, in part, to address these issues.

A central theme of the DARPA SEC program is to develop software based control technologies that use dynamic information about the controlled system to adapt, in real-time, to changes in the subsystems and its operating environment. This software should be flexible to reconfiguration, allocate sensor capability and actuator control authority, achieve stable and robust operation across complex mode transitions, integrate and coordinate subsystem operations and enable large-scale distribution of control [1]. The Open Controls Platform (OCP) [2], being developed by Boeing St. Louis, Honeywell Laboratories, Georgia Institute of Technology and the Oregon Graduate Institute, under the DARPA SEC program provides a software infrastructure to enable control engineers to work seamlessly, in real-time and simulation-time, within a distributed, control environment. The OCP software is built upon RT-CORBA and is an extension to the Bold Stroke software architecture developed by Boeing St. Louis to support aircraft avionic system integration. The OCP is middle-ware that consists of a set of services that allow multiple processes running on one or more machines interacting across a network.

The initial implementation of the OCP required users to have significant computer science background to implement a controller, which most control engineers do not have. Iteration of design and implementation through interaction of control designers and embedded system specialists can result in long product development time. The SEC program was designed to bridge the gap between control science and computer science so that control engineers can take their design from paper to hardware without having to deal with the details of implementation. Since the OCP is the proposed solution to bridge this gap, it is obvious that the success of the OCP in the controls community is strongly governed by level of its integration with MATLAB, a popular control design software. In this chapter we demonstrate the integration of MATLAB with the OCP with the help of three examples, based on flight control design for an F-16 aircraft model.

The chapter is organised as follows. A brief description of the OCP is provided in section 2, followed by details of the F-16 model used in the examples. Three examples present different levels of integration of control algorithm within the OCP environment. A discussion follows the examples on a proposed C++ code generation tool in MATLAB that will translate the simulation setup in SIMULINK to the OCP environment. Issues involved in creating asynchronous realtime systems from SIMULINK are also discussed. The chapter ends with a section on conclusions.

2 What is the OCP?

The OCP is an object oriented software infrastructure that allows seamless integration of cross-platform software and hardware components in any control system architecture. It is designed to facilitate analysis, development and testing of control algorithms in a distributed computing framework. The current application focus of the OCP is in the domain of uninhabited aerial vehicles (UAVs) flocks.

The OCP is a middleware that is based on realtime common object request broker architecture (RT-CORBA). A middleware is a connectivity software that consists of a set of services, allowing multiple processes running on one or more machines to interact across a network. In the computer science jargon, a middleware can take on the following different forms:

- *Transaction processing (TP) monitors* - which provide tools and an environment for developing and deploying distributed applications.

- *Remote Procedure Call (RPCs)* - which enable the logic of an application to be distributed across the network. Program logic on remote systems can be executed as simply as calling a local routine.

- *Message-Oriented Middleware (MOM)*, which provide program-to-program data exchange, enabling the creation of distributed applications. MOM is analogous to email in the sense it is asynchronous and requires the recipients of messages to interpret their meaning and to take appropriate action.

- *Object Request Brokers (ORBs)* - which enable the objects that comprise an application to be distributed and shared across heterogeneous networks.

The essential feature of the OCP, from a user point of view, is an OCP component. For control systems, an OCP component is a subsystem that is realised in the distributed computing framework. It could either be a pure software entity or a hardware entity with necessary software interface with the OCP. Interaction of the various OCP components results in the functioning of the designed control system. Inter-component data is transfered via the publish-subscribe message passing model. When multiple applications need to receive the same data or message, publish-subscribe messaging is used. The central concept in a publish-subscribe messaging system is the topic. Multiple publishers may send messages to a topic, and all subscribers to that topic receive all the messages sent to that topic.. From control systems point of view, a topic would be the output port of a subsystem or an OCP component. All components, whose input ports are connected to this output port, subscribe to this topic. Hence, whenever there is data available at the output, it reaches the input of those who subscribe to it. This model is extremely useful when a group of applications want to notify each other of a particular occurrence.

3 F-16 aircraft model

The nonlinear F-16 model used in this chapter is available in [3], which we shall represent as

$$\dot{x} = f(x, u) \tag{1}$$

The mathematical model uses the wind-tunnel data from the NASA-Langley wind-tunnel tests on a scale model of an F-16 aircraft [4]. The aerodynamic data is valid up to Mach 0.6, angle of attack range of $-10^\circ \leq \alpha \leq 45^\circ$ and side slip angle range $-30^\circ \leq \beta \leq 30^\circ$. The wind-tunnel tests were conducted on sufficiently close points to capture the nonlinear behavior of the aerodynamic force and moment co-efficient. Aerodynamic data for the intermediate points are linearly interpolated. The states $x \in \mathcal{R}^{12}$ and controls $u \in \mathcal{R}^4$ in the model are defined as

npos = North position (ft)	T = Thrust (lb)
epos = East position (ft)	δ_e = Elevator deflection (deg)
h = Altitude(ft)	δ_a = Aileron deflection (deg)
ϕ = Bank angle (rad)	δ_r = Rudder deflection (deg)
θ = Pitch angle (rad)	
ψ = Yaw angle (rad)	
V = Velocity(ft/s)	
α = Angle of attack (rad)	
β = Side slip angle (rad)	
p = Roll rate (rad/s)	
q = Pitch rate (rad/s)	
r = Yaw rate (rad/s)	

Actuators for the control surfaces and engine are modelled as first order systems, details of which are available in [3] and [5]. This model is an example of a high performance UAV that will be used to demonstrate the capabilities of the OCP. The F-16 model is available as a SIMULINK block, as an ANSI C function, as a C++ object and as an OCP component. The SIMULINK implmentation is available with OCP distribution 1.0. It serves as the baseline fixed wing UAV model for the SEC program.

4 Integration of MATLAB with OCP

It is clear, for the Open Controls Platform to be useful for the controls community, it must be well integrated with MATLAB. Most control engineers test new control algorithms in SIMULINK environment. The convenience of rewiring interconnections, adding, deleting and modifying blocks within the

MATLAB environment, has made SIMULINK a popular simulation tool.

Recent versions of MATLAB have features that enable control engineers to design control algorithms in SIMULINK and implement it on off-the-shelf DSP boards from MOTOROLA and Texas Instruments. Several third party vendors like dSPACE and RTSim also supplement the realtime implementation of SIMULINK. MATLAB's Realtime Workshop toolbox provides tools that can convert SIMULINK subsystem definition into intermediate ANSI C code. The C code can subsequently be compiled to run in the some of the off-the-shelf DSP boards. Thus MATLAB/SIMULINK provides not only an environment for system level design, it also has necessary infrastructure for rapid realtime prototyping.

Hence, its imperative to integrate SIMULINK with the OCP, and take advantage of such an established and popular software package.The following subsections describe in detail the exact nature of MATLAB and OCP integration demonstrated in the three examples.

4.1 Linear Quadratic Regulation of F-16

The first interface of MATLAB with the OCP was developed using client-server architecture. A linear quadratic regulation (LQR) of the F-16 model was demonstrated. The entire setup was created in SIMULINK as shown in figure(1). A six state model, describing the longitudinal motion, of F-16 was used to demonstrate the MATLAB-OCP integration. The LQR controller was designed to regulate state perturbations to zero, using thrust and elevator control. This example was developed for the initial OCP version 0.1 to demonstrate the capability of integrating MATLAB/SIMULINK with the OCP middleware.

In the setup shown in figure(1), the block labelled *Remote Controller* is the linear quadratic regulator. It is implemented as a `mex` function written in C which interfaces with the OCP. Prior to starting the simulation, a MATLAB process has to be started, which behaves as a LQR server. The LQR server is like any other server program in the computer science context. Typically, its a program that is *bound* to some port of a computer and waits for some client program to connect. It is activated when a client connects to the designated port to establish a connection. The LQR server is designed to read in an

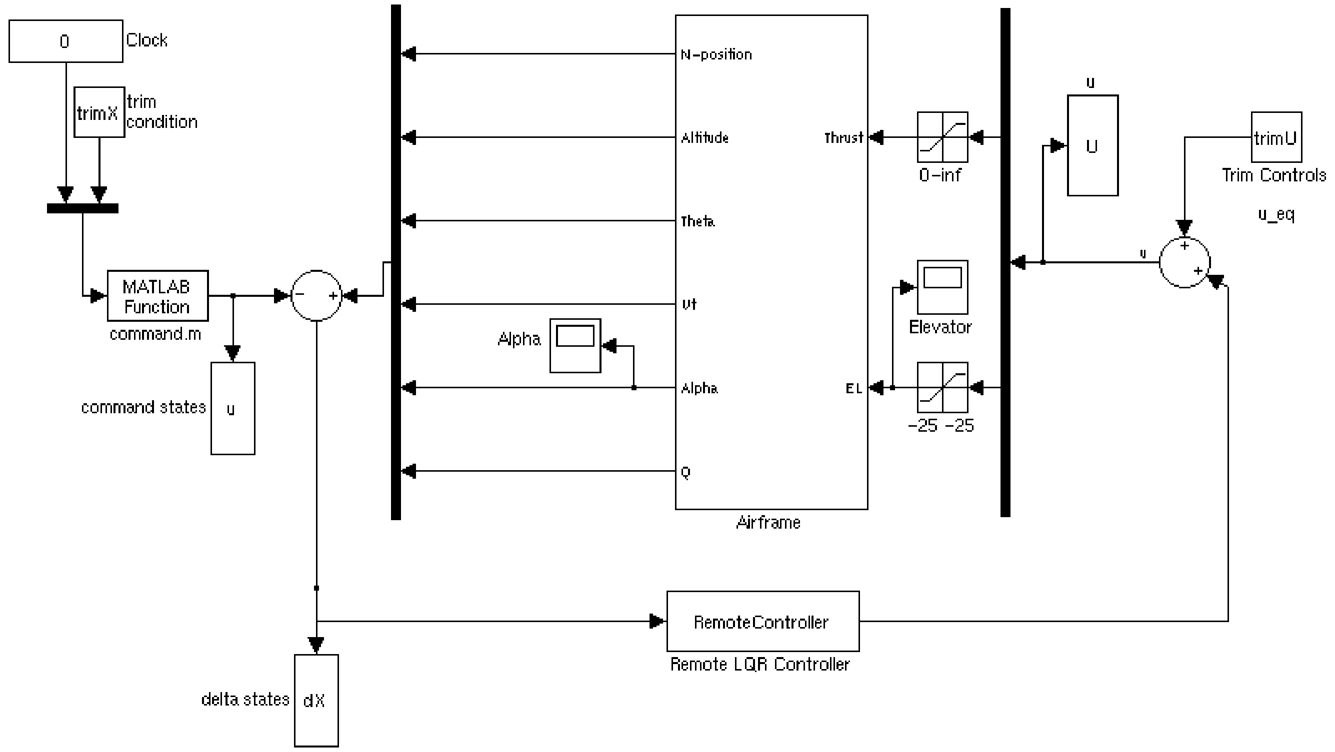

Figure 1: LQG Control in SIMULINK-OCP framework.

array of numeric data, representing the state vector x, multiply it with a matrix K to produce a control vector $u = Kx$, and write it to the port for the client to read. This is illustrated in figure(2). The LQG controller is written completely in MATLAB. A set of socket interfaces were written to enable basic network programming in MATLAB.

The client, a OCP component sitting inside the SIMULINK block labelled *Remote Controller*, connects to the MATLAB based LQR server. The LQR server can be configured with the controller K during startup. The OCP client connects to the LQR server and establishes a connection. At every time step when SIMULINK invokes the *Remote Controller*, the OCP client sends the input vector signal to the LQR server. The LQR server generates the control signal and sends it to the OCP client. The OCP client then writes it to the output port of *Remote Controller*. Note that this form of message passing is very different from the publish-subscribe model of message passing. The *Remote Controller* OCP component essentially had no active role in the control of the aircraft. It simply served as a socket connection to the MATLAB process.

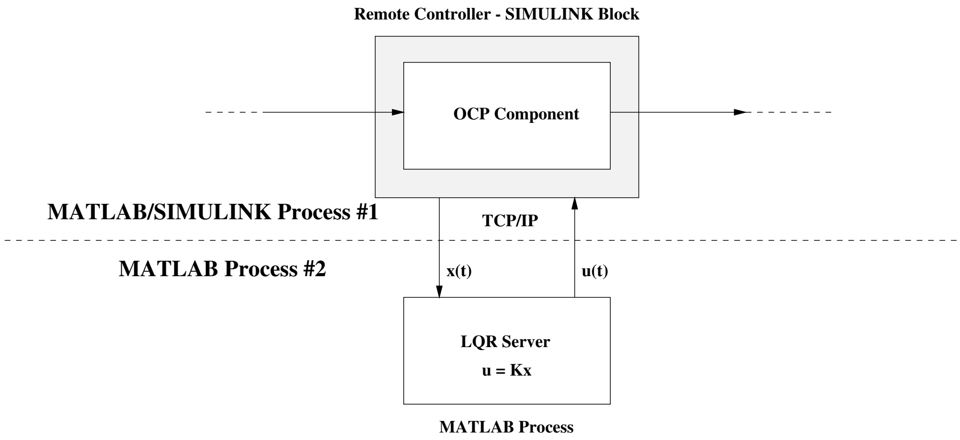

Figure 2: IPC of MATLAB Processes over TCP/IP

There are obvious drawbacks in this form of interface. This client-server architecture has serious potential for deadlock condition. In this setup, once the client sends the state vector to the LQR server, it waits for the server to respond with the control signal. On the other hand, the server program is expecting the client to send the state vector. If for some reason, the data sent by the client doesn't reach the server, then both client and server wait for each other to transmit data and a deadlock situation arises. This can be fixed by implementing the concept of timeout and retrying, but such an approach is unsuitable for real-time implementation.

It is important to note that SIMULINK drives the simulation. The OCP manager is invoked when SIMULINK executes the *Remote Controller* block. There is only one OCP component in this simulation and it is triggered by SIMULINK at fixed intervals of time.

This example demonstrates the initial effort in integrating MATLAB with the OCP. The OCP acts as a sophisticated socket connection between two MATLAB processes. The simulation is driven by SIMULINK, hence it is highly synchronous in execution. The ability of the OCP to execute components asynchronously is not exploited in this example. This example was designed to demonstrate a socket based client-server interface between MATLAB and the OCP.

4.2 LPV Regulation of F-16

The second control application of the OCP integrates MATLAB as a visualisation package, where SIMULINK scopes are used to visualise F-16 state trajectories. A linear parameter varying (LPV) regulator was designed [5] to regulate the perturbation states of the twelve state F-16 model. The control variables used were thrust, elevator, aileron and rudder. The interconnection needed to implement the LPV controller was developed entirely in C++. Converting a SIMULINK model file to C++ manually can be nontrivial. Hence, to simplify this conversion, a C++ object for linear systems, LPV systems, numerical integration modules and the full nonlinear F-16 model was developed.

The simulation setup in the OCP framework is shown in figure(3). There are two OCP components in this example, namely the F-16 component and the LPV controller component. Both of these components are implemented using the object oriented design philosophy of software engineering. From the application point of view, each of the two OCP components consist of a `main()`{...} function and a `push()`{...} function. The `main()` function, like any ANSI C program, is the entry point of the code. The `push()` function is the event handling code. In this example, the F-16 component subscribes to the output of the LPV controller and the LPV component subscribes to the output of the F-16 component. The `push()` function acts as a event handling routine that reads in the signal from input port, computes the output signal and publishes it. Note that this function is only called when an event occurs.

To explain the inner workings of this example, let us assume that the executable code for F-16 is `a.out.F16` and for LPV controller is `a.out.LPV`. To start the simulation, these two programs will have to be started in two different processes or `xterms`. The order of invocation is not important. When both the components have been started, each is waiting for the other to publish a signal. Neither can publish any signal because the necessary input is missing. This results in a deadlock. This can be solved by first computing the control signal $U(t)$ from the initial state vector X_0 of the F-16 model. The `push()` function in the F-16 component can't publish this signal as it can only be invoked when the LPV controller generates its output and triggers an event. Thus, the initial state vector X_0 is published by the `main()` function of the F-16 component during startup. This published signal is captured by the controller and it computes and publishes the control signal $U(t)$. The control signal is captured by the F-16 component and the system states are updated, and subsequently published. The alternating publication of F-16 states and control signal results in the LPV regulation of the F-16 model. With reference to figure(3), the exact sequence of code execution is explained below,

1. LPV regulator component is started. It waits for the F-16 component to be activated and publish the initial state vector.

2. F-16 component is started, it

   - connects to MATLAB server through TCP/IP sockets, uses handshaking mode of data communication

   - connects to LPV regulator through publish-subscribe messaging and exchanges acknowledgement messages

   - sends the initial state vector of the F-16 model

3. LPV captures the state vector, computes control signal and publishes it.

4. F-16 captures the control signal, updates states and publishes the state vector. It also sends the state vector to the MATLAB server for trajectory visualisation. Note that the data flow from F-16 component to MATLAB process is not through publish-subscribe but through TCP/IP compliant sockets.

The last two steps are repeated until the simulation is over. The F-16 component then sends a termination signal to the LPV controller and both the processes terminate.
This example demonstrates the first publish-subscribe based flight control of the F-16 model using the OCP. In this example, MATLAB plays no role in controlling the airplane. It is used as a visualisation package to plot the aircraft state trajectories in realtime. Development of these simple OCP components revealed some of the complexities involved in using the OCP. The LPV controller was first tested in SIMULINK environment and then ported to the OCP environment. The convenience of MATLAB tools and SIMULINK blocks was missing when everything had to be converted to C++. Since interfacing to the OCP required all SIMULINK blocks to be recreated as C++ objects, development and testing of

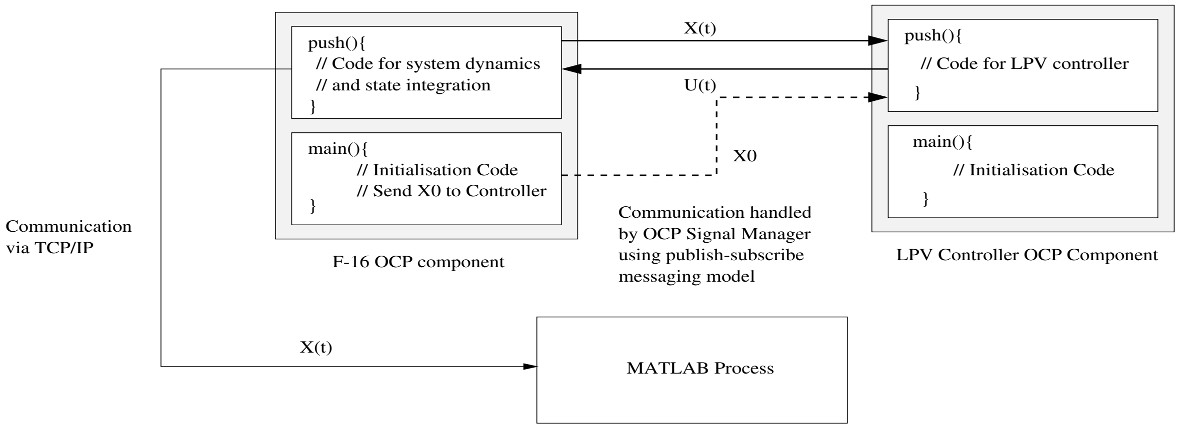

Figure 3: LPV Regulation in the OCP Framework

these objects was a time consuming endeavour. It was also revealed that C++ programming knowledge alone is not sufficient to successfully develop an OCP component. Knowledge of the operating system and OCP internals are also necessary. We discovered that improper file I/O in the program can potentially cause race conditions that halt the computer. A race condition occurs when two processes try to access a shared resource and neither can. This race condition was resolved by changing the sequence of code execution.

Thus with this example, we were able to highlight the need for tighter links between MATLAB/SIMULINK and the OCP. This will not only reduce the simulation setup time, but will also improve reliability of results. It was also clear that a code generation tool was necessary to make the internals of the OCP transparent to the users of OCP. We were also able to demonstrate how an external visualisation package can be linked to the OCP via socket connection.

4.3 Waypoint Tracking Problem

The third example is a waypoint tracking problem. In this problem, we are interested in having the F-16 aircraft model reach co-ordinates in space at specific times and states of the aircraft. Waypoint tracking is implemented as an outer-loop command generator and an inner-loop tracking controller. The aircraft inner-loop controller consists of two parts, the lateral controller and the longitudinal controller. The lateral controller is a classical controller given in [4] and the longitudinal controller is a LPV controller designed to track velocity(V) and flight path angle (γ) commands. The outer-loop controller, developed by Honeywell Laboratories researchers, generates velocity $V(t)$, climb angle $\gamma(t)$ and roll rate $p(t)$ reference signals on the basis of specified waypoints and the current position of the airplane. The inner-loop controller was designed to track these signals. The combined effect of the inner and outer-loop controller flies the airplane to the specified way-points at the specified times. Implementation of the waypoint tracking framework is shown in figure(4). In this example, a switch was added to allow the waypoint command generator to be disconnected and have a joystick generate the required inner-loop commands. We use a three button analog joystick to generate $[V(t), \gamma(t), p(t)]$ reference signals. It is possible to select the waypoint generator or the joystick by clicking the manual switch in the SIMULINK diagram.

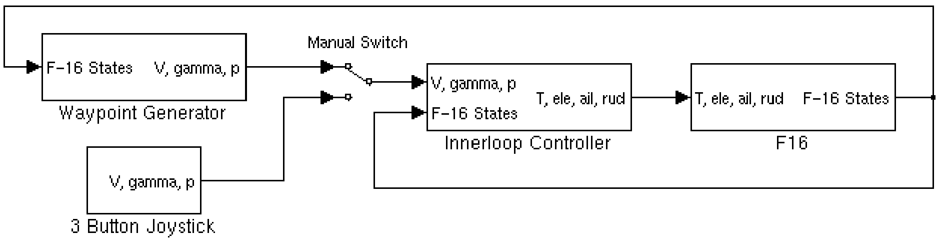

Figure 4: Waypoint Tracking Setup

The outer-loop controller is shown in figure(5). It involves two sub-systems, the way-point loop and the flight path loop. The way-point loop determines where the aircraft is located and estimates the velocity command vector, $[\chi(t), V(t), \gamma(t)]$. The velocity command vector is converted to reference signal, $r(t) = [V(t), \gamma(t), p(t)]$ by the flight path loop. There is also a way-point manager, shown in figure(6), that decides when to update the current way-point. The waypoint generator requires three waypoints to determine the reference signal.

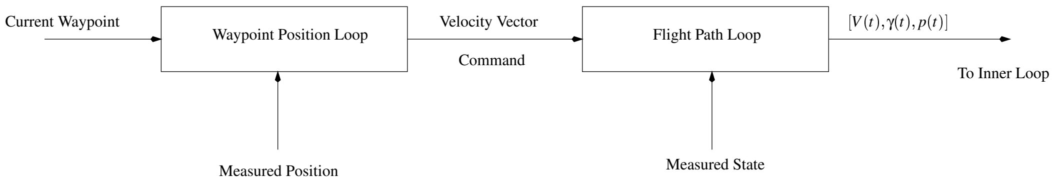

Figure 5: Honeywell outer-loop controller

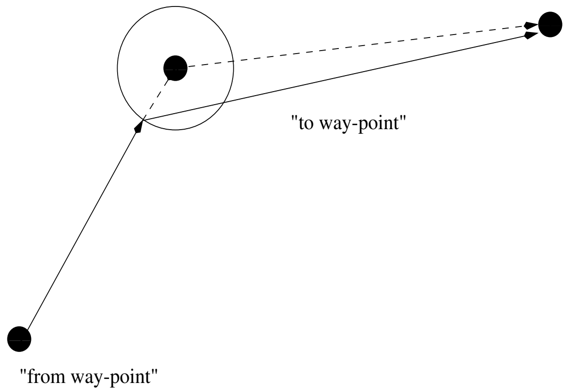

Figure 6: Honeywell way-point generator

The equations of motion used to determine the position of the aircraft are

$$\begin{bmatrix} \dot{\xi} \\ \dot{\eta} \\ \dot{h} \end{bmatrix} = \begin{bmatrix} V cos(\gamma)\, cos(\chi) \\ V cos(\gamma)\, sin(\chi) \\ sin(\gamma) \end{bmatrix} \tag{2}$$

where ξ is the east position, η is the north position and h is the altitude of the aircraft. From equation(2), we can obtain equations for χ, V and γ as

$$\begin{bmatrix} \chi \\ V \\ \gamma \end{bmatrix} = \begin{bmatrix} \frac{\Delta\eta}{\Delta\xi} \\ \frac{\sqrt{(\Delta\xi)^2+(\Delta\eta)^2+(\Delta h)^2}}{\Delta t} \\ sin^{-1}\left(\frac{\Delta h}{\sqrt{(\Delta\xi)^2+(\Delta\eta)^2+(\Delta h)^2}}\right) \end{bmatrix} \tag{3}$$

Dynamic inversion [6] is used to compute thrust, velocity and roll rate which will fly the commanded velocity vector. The aircraft model is assumed to be a linear model given by

$$\dot{x} = Ax + Bu \tag{4}$$

$$y = Cx \tag{5}$$

and the control law is given by

$$\dot{y} = C\tilde{A}x + C\tilde{B}u \tag{6}$$
$$u = (C\tilde{B})^{-1}\left[\dot{y}^{des} - C\tilde{A}x\right] \tag{7}$$

This method is heavily model based. $\tilde{A}$ and $\tilde{B}$ are models of the dynamics embedded in the control law given in equation(7). The nonlinear model of the F-16 aircraft is linearised at every timestep during simulation and used in dynamic inversion to compute the reference signals.

The longitudinal axis LPV controller, was designed to track the $V(t)$ and $\gamma(t)$ reference signals. It also decouples the tracking, which provides two degrees of freedom in controlling the longitudinal flight path. The decoupling is essential because the outer loop is expected to generate commands where γ is fixed and velocity changes or vice versa. The controller was designed using model matching technique. The ideal models for velocity and flight path angle tracking are given as

$$T_{ideal,V} = \frac{0.16}{s^2 + 0.8s + 0.16} \tag{8}$$
$$T_{ideal,\gamma} = \frac{-\frac{1}{3}s + 1}{s^2 + 2s + 1} \tag{9}$$

The LPV controller is scheduled on velocity and is valid for velocity ranging from 400 ft/s to 800 ft/s at an altitude of 10,000 ft.

This example is similar to the first example in the sense that the simulation is driven by SIMULINK. Here the plant model is an OCP component. SIMULINK is interfaced with the OCP via `S-functions`. In this example, communication between OCP and SIMULINK is achieved via publish-subscribe messaging. This differs from the first example where socket based data communication is used. There is a minimal amount of integration with the OCP since there is only a single OCP component present. Deeper integration of the OCP with SIMULINK could have been achieved if all the subsystems in figure(4) were converted to OCP components. Since the current version of the OCP allows components to be only defined in C++, converting the complicated interconnections in the subsystems would have required a very large development and validation time. Hence, it was avoided. This once again highlights the requirement of a code generation tool that will convert SIMULINK subsystems to OCP components.

One of the key element in this example is the analog joystick. The joystick was included to enable tracking of human generated reference signals. Since the joystick is an input device, inclusion of this hardware did not impose any realtime constraints on code execution. The joystick was implemented as a SIMULINK block using DirectX APIs in `S-functions`. The waypoint tracking example also integrates an external visualisation software, AVDS [7] for realtime visualisation of F-16 state trajectories.

4.4 Summary

The three examples presented in this section demonstrate three different ways of integrating MATLAB with the OCP. Each example depicts a more complex flight control scenario than the previous. However, none of these examples fully demonstrate the capabilities of the OCP. The OCP components in the three examples execute in a highly synchronised manner. The ability of the middleware to handle asynchronous events like mode switching and other aspects of reconfigurability were not exemplified. Also, these examples were executed in a simulated environment. Hence, issues regarding schedulability and realtime CPU task allocation were not considered during both development and implementation phases. These simulations were initiated to explore the possible ways of integrating MATLAB with the OCP. The experience obtained from developing these initial OCP based examples, helped us to recognise scopes for improvement and avenues for further research that will make the OCP more user friendly to control engineers.

5 Simulink to OCP Components

The current version of the OCP provides a tool called *sim2ocp* that helps users to create OCP components from SIMULINK diagrams. Currently, only the data flow specification of the simulation setup can be specified by the SIMULINK model file. The code generated by *sim2ocp* only handles the I/O interface of the components. The internal behaviour of each component has to be manually coded. This is not trivial for most control engineers as it requires C++, CORBA and operating system level knowledge.

The *sim2ocp* GUI has six tabs or pages where necessary are entered. The different pages and the information entered are explained as follows.

- **Import SIMULINK Model File** - This requires the user to enter the SIMULINK model filename. The data flow information is obtained from the interconnection of the input and output ports.

- **Signal Definitions** - The names of subsystems and their ports are determined by *sim2ocp* from the SIMULINK diagram. The user however has to define the signal vector that connects these ports. Signal vectors are created with the help of `struct` data structure available in ANSI C. Each member of this user defined `struct`, defines an element of the signal vector. For example, for our F-16 example, we would need to define a signal vector that connects the output port of the controller to the input port of the F-16 component. This signal vector would contain thrust, elevator, rudder and aileron control variables. Thus we would need to define a `struct` for this signal as follows

```
typedef struct {
        double Thrust;
        double Elevator;
        double Rudder;
        double Aileron;
        } ControllerOutput;
```

- **Ports** - Once the necessary signals have been created, the signals need to be identified with the ports of the subsystems. The *sim2ocp* tool allows reusability of signal data structure wherever possible. This page also allows the user to specify scheduling information like criticality, execution rates, worst case execution time, aperiodic deadline and a choice to execute code at periodic interval. The periodic interval could be based on software or hardware timer.

- **Components** - The SIMULINK block diagram entered in page one, could have an interconnection of more than two subsystems. The user has the freedom to specify the number of OCP components generated and allocate the number of subsystems that will be contained in each component. Each OCP component behaves as an independent process, within which several subsystems can execute. Thus, the importance of scheduling comes into the picture when a single process has to manage the execution of more than one subsystem. In this page, the user can also choose to delay the execution of a subsystem until all its input dependencies have been satisfied.

- **Configuration States** - This page allows the user to specify different configuration states for mode switching scenarios. It is possible to create different configuration states where certain ports are activated or deactivated, criticality of the subsystems modified and execution rates changed.

- **Finish** - This is the last page of the GUI. Once all the information has been entered, it is saved in a *Component Information File*. The C++ code for OCP components and specific I/O handling routines are generated by a `perl` script called `genopc.pl` based on the *Component Information File*..

The `perl` script generates Microsoft Visual C++ project files along with the code for the OCP components. The generated code, however is not ready to be compiled at this point. The code generator only generates the realtime asynchronous I/O interfaces. This tool makes the I/O interface transparent, but is still not very useful for control engineers. The code generator doesn't generate the subsystem definition. The generated code has regions where the user has to manually code the subsystem definition in C++. This, as we mentioned earler, is often nontrivial for complex systems.

5.1 Proposed modifications to `sim2ocp`

We propose an approach that will use SIMULINK model files not only to extract data flow specifications, but also the internal behaviour of the components. The same SIMULINK diagram that was used to test the simulation can be used to generate C++ code for OCP components. Since our proposition is based strongly on code generation, it is important to point out that there exists concern about reliability and limits of performance of code generation technology. We feel that as this technology matures, we will begin to trust the generated code as we currently trust the assembly code generated by C or FORTRAN compilers .

We explain our proposed modification to *sim2ocp* by using the third example of OCP application, i.e the waypoint tracking problem. Building a simulation or actually implementing a control algorithm in the OCP framework means to integrate different hardware and software components into one interconnection. Thus, to cast the waypoint tracking problem into the OCP framework, we would require to break down the problem into separate units or systems that will communicate with each other via the OCP.

The waypoint tracking setup shown in figure(4) is divided into three sub-systems, the F-16 model, the innerloop controller and the outerloop controller. The joystick is eliminated for this discussion. These subsystems will be converted to OCP components, which are assumed to be distributed in nature, i.e there is no direct data link between any two components. Data flow from one component to another would be managed by the OCP through publish and subscribe model of message passing. Once the system interconnection is designed and tested in the MATLAB/SIMULINK environment, it can be ported to the OCP environment using the MATLAB command `sim2ocp`. The original *sim2ocp* is a GUI based tool written in Visual Basic. We propose a similar tool in MATLAB that incorporates the suggested modifications.

5.2 Inner Workings of `sim2ocp`

The code generation tool is invoked with the command

```
sim2ocp(mdlFile,OPTION);
```

where `mdlFile` is a MATLAB string containing the SIMULINK model filename and `OPTION` is a MATLAB data structure that has necessary parameters to customise code generation. The SIMULINK file in the argument list is assumed to be an interconnection of subsystems only. Including any other SIMULINK block in the root diagram will result in an error. Each SIMULINK subsystem is converted into an OCP component. Issues regarding realtime scheduling and mode switching is not handled by this version of MATLAB `sim2ocp`.

This tool is primarily being developed to explore possibilities for automatic extraction of subsystem definition from SIMULINK `.mdl` files so that manual coding in C++ can be avoided. The goal is to automatically synthesise OCP components from SIMULINK diagrams and obtain simulation results similar to those obtained from SIMULINK. The stress of this effort is on automated and accurate synthesis of requisite software interface that will allow seamless transition from the SIMULINK world to the OCP environment.

SIMULINK imposes certain priority conditions on the order in which subsystems are evaluated to help in handling algebraic loops. The `sim2ocp` tool will impose similar conditions on the execution order of the OCP components in terms of input dependencies. It is not necessary that the OCP components execute in the same order as SIMULINK blocks. We impose this order because we will be validating the accuracy of code generation by comparing the simulation results with those of SIMULINK, and the order of execution affects the simulation result.

The `sim2ocp` tool essentially has a parser and a code generator. The parser extracts the following information from the SIMULINK diagram.

- **Subsystem name** - the subsystem name defines the name of the OCP component.
- **I/O properties** - the number of inputs, outputs are determined from the model file. It also determines if the system has direct feed through, i.e if the current value of input is needed to compute the current value of output. It creates the required number of signal vectors to interconnect the subsystems.
- **Interconnection** - the data flow specification is parsed from the SIMULINK model file.

Once `sim2ocp` identifies the subsystems and determines the I/O details, it creates a separate `.mdl` file for each subsystem. For the waypoint tracking example, it will create three `.mdl` files. The code generator incorporates the subsystem definition in addition to the OCP I/O interfaces. The mapping of input to output is obtained either by evaluating SIMULINK subsystems via the MATLAB engine APIs or by converting the subsystems into ANSI C code with the help of Real-Time Workshop toolbox of MATLAB. Details of the SIMULINK wrapper object is as follows,

- The wrapper can make use of the MATLAB Engine API to evaluate SIMULINK subsystems. It is possible to call a SIMULINK model file from MATLAB, as a function call to obtain either the output or the derivatives of the model. The MATLAB Engine API extends this capability to C++ code. The MATLAB engine is a minimal MATLAB process that communicates with Unix/Windows processes using pipes/ActiveX. It is possible to manipulate this virtual MATLAB workspace, perform computations and extract output using C++ code. The wrapper exploits this capability to evaluate SIMULINK model files. Since MATLAB is used to map input to output, it is time consuming. This implementation is not expected to meet realtime deadlines. It should primarily be used for rapid testing in the early stages of design.

- It is also possible to convert a SIMULINK model file into pure C code using the Real-Time Workshop toolbox of MATLAB. In this case, the same SIMULINK wrapper calls the generated C functions instead of the MATLAB engine. This form of implementation is expected to be atleast ten times faster than the MATLAB engine based implementation.

- The wrapper is equipped with numerical integration algorithms to update both continuous and discrete time states.

- The wrapper has the ability to change simulation parameters like integration time step, order of numerical integration algorithm, etc. as the simulation takes place. This allows dynamic reconfigurability of computational load for a given OCP component.

The main OCP event handler code, that is notified when a publisher triggers an event, simply passes the input vector to the wrapper. The wrapper computes the output and updates states, if any. All simulation related book keeping is internally handled by the wrapper. The tool `sim2ocp` generates `.c` and `.h` files and builds a operating system executable version of the components. It can also create `mex` files for the desired components. The `mex` versions of the components can be invoked from either MATLAB command prompt or used as SIMULINK blocks. The overall execution speed is expected to be slower with `mex` wrapper incorporated.

The proposed tool makes the transition from SIMULINK to OCP very transparent. It speeds up the development time of OCP components and serves as an early rapid application development (RAD) tool for the Open Controls Platform middleware. Issues regarding real-time scheduling and mode switching will be addressed in the future versions of this tool. The protocols for handling these issues are still under development. The OCP development team, in collaboration with other SEC technology developers, is currently addressing these issues to come up with appropriate scheduling and realtime analysis tools.

6 Asynchronous Systems and SIMULINK Models

The SIMULINK diagram defines a dataflow specification of the system being simulated. Data flow specifications can be used to automatically synthesise asynchronous systems [8]. The dataflow model, obtained from SIMULINK diagram, represents many asynchronous concurrent computation events [9], where each event corresponds to the activation of a component. A component becomes active when all the inputs it requires are available. Thus in dataflow execution, many components may be ready to execute simultaneously, resulting in several asynchronous concurrent computation events.

In SIMULINK, subsystems are executed sequentially [10]. The order of execution is determined during the initialisation phase of the simulation. A list is created such that any block with direct feedthrough is not updated until the blocks driving its inputs are updated. Such a list cannot be created if there are algebraic loops in the interconnection. Typically, algebraic loops are created when the output of a block with direct feedthrough is fed to its input, directly, or by a feedback path through other blocks with direct feed through. Simulation in SIMULINK is a two step process. First, output of each block is computed in the order determined during initialisation. Then in the second pass, the system states are updated based on their current state and derivatives.

Once the SIMULINK subsystems are converted to OCP components and implemented within the OCP, their execution will occur in an asynchronous manner. Generally, the order of their execution in this environment may not be the same as that in SIMULINK. This difference in order of execution could lead to results that are different from that obtained from SIMULINK, though not necessarily erroneous.

It is clear that the simulation setup itself imposes certain synchronisation of components. For instance, in the waypoint tracking example, the activation of the waypoint generator component is dependent on the availability of sensor data of F-16. The sampling rate of all the sensors may not be the same, hence the dependencies of the waypoint generator may not be satisfied all at once. Its activation has to be delayed until all the dependencies have been satisfied. SIMULINK satisfies this constraint when it defines the order in which the blocks are evaluated. In the asynchronous environment, this has to be explicitly satisfied. However, it may not be always necessary to delay the activation until all the dependencies are satisfied. For certain applications, it may be acceptable to use the input from previous update to compute the current output.

Realtime deadlines for software elements, can be hard or soft deadlines. However, if there are hardware elements in the interconnection, then the realtime deadlines are hard deadlines. A hardware element will demand its input buffer to be filled with valid data at every sampling time. It is not possible to delay the execution of such an element until all its input dependencies have been satisfied. In case of violation of realtime deadlines, the system either propagates with partially current input signals or the sources of the input dependencies approximate their outputs to meet the realtime deadline. These errors will affect the overall performance of the control system. Simulating in SIMULINK do not reveal these issues at all. Thus the migration, from SIMULINK models to real life asynchronous systems, is a large step.

7 Conclusions

A central theme of the DARPA SEC program is to use dynamic information about the controlled system to adapt, in real-time, to changes in the systems and its operating environment. A key enabling technology is the OCP middle-ware software. The OCP services will enable dynamic control algorithms to handle of complex mode transitions, integrated and coordinates subsystem operations and enable large-scale distribution of control. This chapter describes software tools developed to link the Matlab/Simulink environment and the OCP. These tools are key to allowing control engineers to seamlessly implement control algorithms within the OCP framework. To fully realize the benefits of the OCP in future military and commercial systems, the controls and computer science communities need to establish close ties and aggressively address issues at the real-time control/software interface.

References

[1] "DARPA - Software Enabled Control," http://www.darpa.mil/ito/research/sec/index.html.

[2] Paunicka, J., Mendel, B., and Corman, D., "The OCP - An Open Middleware Solution for Embedded Systems," *American Control Conference*, 2001.

[3] Stevens, B. and Lewis, F., *Aircraft Control and Simulation*, Wiley-Interscience Publication, John Wiley & Sons, Inc., 1992.

[4] Nguyen, L., Ogburn, M., Gilbert, W., Kibler, K., Brown, P., and Deal, P., "Simulator Study of Stall/Post-Stall Characteristics of a Fighter Airplane with Relaxed Longitudinal Static Stability," Tech. Rep. 1538, NASA Langley Research Center, Hampton, Virginia, 1979.

[5] Bhattacharya, R., Balas, G., Kaya, M., and Packard, A., "Nonlinear Receding Horizon Control of F-16 Aircraft," *American Control Conference*, Vol. 1, 2001, pp. 3102–3105.

[6] Bugajski, D. J. and Enns, D. F., "Nonlinear Control Law with Application to High Angle-of-Attack Flight," *Journal of Guidance, Control and Dynamics*, Vol. May-June, 1992, pp. 761–767.

[7] "Aviator Visual Design Simulator," http://www.rassimtech.com/.

[8] Wuu, T.-Y. and Vrudhula, S. B. K., "Synthesis of Asynchronous Systems from Data Flow Specifications," Tech. rep., Information Sciences Insitute, University of Southern California, ISI No. RR-93-366, NTIS No. ADA-278687, Dec. 1993, http://citeseer.nj.nec.com/wuu93synthesi.html.

[9] Najjary, W. A., Lee, E. A., and Gao, G. R., "Advances in the Dataflow Computational Model," http://citeseer.nj.nec.com/najjary99advance.html.

[10] "SIMULINK Manual - Using Simulink," http://www.mathworks.com.