Proceedings of the
44th IEEE Conference on Decision and Control, and
the European Control Conference 2005
Seville, Spain, December 12-15, 2005

Application Programming Interface for Real-Time Receding Horizon Control

Tamás Keviczky, Andrew Packard, Oreste R. Natale, Gary J. Balas

Abstract— **An application programming interface (API) was developed to support implementation of receding horizon control (RHC) schemes within the Open Control Platform (OCP) real-time software environment. The basic philosophy and process timing architecture is presented, along with details of a prototype implementation of a quadratic programming based generic RHC scheme. The API framework relies on a real-time software infrastructure and provides a high-level interface to control engineers, which simplifies the embedded control design and implementation process significantly. The RHC API was successfully flight tested on a full-scale aircraft in the DARPA-sponsored Software Enabled Control program final demonstration experiment.**

I. INTRODUCTION

The growing complexity of control applications requires a software infrastructure that supports the developer in leveraging from inter-process communication, operating systems, the implementation details of tasks scheduling, and low level device control software in a seamless manner [1]. This enables the developer to concentrate all his design efforts on the overall system behavior. One of the ultimate goals of the DARPA Software Enabled Control (SEC) program [2] was the development of the Open Control Platform (OCP), which is a software technology supporting real-time distributed control application development and implementation [3]. The Receding Horizon Control Application Programming Interface (RHC API) is a key module within the OCP that was developed to simplify implementation of receding horizon control schemes for control engineers in a real-time software environment.

The paper is structured as follows. Section II gives a general overview of the Open Control Platform, which encompasses the RHC API implementation. Details of the RHC API framework are described in Section III. A brief summary of the DARPA SEC flight test experiment and simulation results are presented in Section IV.

II. OPEN CONTROL PLATFORM OVERVIEW

The Open Control Platform (OCP) provides an open, middleware-enabled software framework and development platform for control engineers and researchers to aid technology demonstration in simulated or actual embedded system platforms [4]. The middleware layer of the OCP provides the software layer isolating the application from the underlying target platform. It provides services for controlling the execution and scheduling of components, inter-component communication, and deployment of application components onto a target system. The embedded system domain of particular interest to the SEC program was that of uninhabited aerial vehicles (UAVs), however the software architecture of the OCP leaves the possibility of applications in other domains open. The OCP provides the following main features:

- An open platform for enabling control research and technology transition.
- Dynamic configuration of components and services.
- A mechanism enabling the transition between execution and fault management modes while maintaining control of the target systems.
- Coordinated control of multiple target systems.
- A software system infrastructure that is isolated from a particular hardware platform or operating system.

A primary motivating factor in implementing a middleware-based architecture is the promise of isolating the application components from the underlying platforms. This allows for a more cost-effective implementation of common software components that could be used across different product lines and re-hosted onto evolving embedded computing platforms.

The OCP middleware is written in C++. It includes an RT CORBA component [5], which leverages the ACE and TAO products developed by the distributed object computing (DOC) research team at Washington University. TAO provides real-time performance extensions to CORBA.

The OCP classifies hard real-time tasks into different rate groups that have to be predefined by the user and must be factors of the fastest rate. The OCP Frame Controller provides a synchronous executive that starts all rate processing. The fastest rate is started once each "minor frame" and slower rates are triggered at appropriate multiples of the minor frame. For the specific example of the DARPA SEC flight demonstration, minor frames were specified to execute at a rate of 20 Hz and a major frame of 2 Hz execution rate was defined to comprise ten minor frames. These major frames will be referred to as computational cycles in Section III-B, which describes the timing architecture of the RHC API implementation for the DARPA SEC testbed. The most important services and components of the OCP are described briefly in the following sections.

A. Middleware services

The OCP provides a set of services in addition to the standard CORBA specification, which gives additional real-time performance enhancements as well as higher level services (e.g. event service and replication utilities). For instance, the OCP's resource management component provides a mechanism for controlling resource in a mode-specific

T. Keviczky and G. J. Balas are with Department of Aerospace Engineering and Mechanics, University of Minnesota, Minneapolis, MN 55455, USA, {keviczky,balas}@aem.umn.edu

A. Packard is with Department of Mechanical Engineering, University of California, Berkeley, CA 94720, USA, pack@me.berkeley.edu

O. R. Natale is with the Dipartimento di Ingegneria, Università degli Studi del Sannio, 82100 Benevento, Italy, o.r.natale@unisannio.it

0-7803-9568-9/05/$20.00 ©2005 IEEE

way. This is an essential element for supporting modes in hybrid systems. The designer specifies quality of service (QoS) information, which is an input into the resource management component to control the run-time execution of the OCP. This component is responsible for partitioning the system resources based on the mode of execution and is an extension of the Honeywell Labs real-time adaptive resource management (RT-ARM) capability [6]. It adjusts rates of execution based on utilization information from the scheduling component and notifies application components when rates have been adapted.

B. Controls API

As mentioned above, the OCP provides several advanced mechanisms such as dynamic scheduling and resource management. To help hide the complexity from the controls designer, the OCP includes a control designer abstraction layer above the RT CORBA implementation. This API allows the designer to focus on familiar tools and terminology while enabling the use of RT CORBA extensions. This help provides a consistent view of the system that is meaningful to the control designer. In order to accomplish this task, the Controls API has been generalized as a combination of a high level description language and a simple programming interface. The designer expresses the characteristics of the system in familiar terms to form a high level description of the system. This description is then processed to populate a component registry which is used by the OCP.

C. A new concept in real-time control: the anytime task

Most control systems built today are resource-limited. This is especially true for embedded control systems in mobile platforms due to constraints of size, weight, space or power. Great effort is expended in engineering solutions that provide jitter-free periodic execution while meeting hard real-time deadlines in systems with high CPU utilization.

Mission-critical command and control is a resource-constrained yet multifunctional enterprise, requiring simultaneous consideration of a variety of activities such as closed-loop control, measurement and estimation, planning, communication, and fault management. On the same computational platform, a large number of tasks must execute.

Anytime or *incremental* algorithms are particularly well suited for implementing tasks that must adapt their resource usage and quality of service [7]. In an anytime algorithm, the quality of the result produced degrades gracefully as the computation time is reduced. In particular, such algorithms may be interrupted anytime and will always have a valid result available. If more computation time is provided, the quality of the result will improve. Examples of applications that could benefit from dynamic resource management range from integrated vehicle health monitoring software to real-time trajectory optimizers that can dynamically replan routes and trajectories of a fleet of UAVs.

The anytime task scheduler makes CPU computing resources available to tasks based on their criticality, computing requirements, and a schedulability analysis. Control tasks execute within their allotted time and are subject to preemption if they attempt to consume more than their allowed resources. The anytime scheduler provides tasks with the information necessary to adapt their computation to the resource available. The application tasks may need to balance the competing demands of deadlines and accuracy, given the resource made available to them. In principle, anytime algorithms can make effective use of any amount of processing time that is available. Anytime tasks are modeled based on the following characteristics:

1) They are continually executing iterative algorithms that are not periodic. Examples include algorithms that continually refine their result (imprecise computation) and that produce new outputs based on new inputs.
2) Computation times and deadlines for each iteration of the algorithm are an order of magnitude larger than the basic periodic rate.
3) The computation time for each iteration is variable and data-dependent. Furthermore, it is possible for the algorithm to adapt its computation time based on the resource allocated.

It is important that anytime algorithms coexist with the periodic tasks in the control system. In order to achieve this coexistence, the anytime task scheduler executes as a periodic task within the overall control system. This periodic task is assumed to run at the system clock rate and can be modeled accordingly for rate monotonic analysis. The scheduler allocates a fixed fraction of the overall CPU time for anytime tasks, and this allocation is then subdivided based on individual anytime tasks. For other OCP features and services, such as detailed information about resource optimization and anytime task scheduling, the reader is referred to [3], [8].

III. THE RHC API

Receding horizon control (RHC) relies on the concept of solving optimal control problems repetitively for a finite future time horizon based on current measurements. The feedback nature of this approach emerges from the policy that only the first value of the control solution is implemented at a given instance, and a new optimization problem is solved over a shifted horizon based on actual measurements at the next time step. The underlying mathematical programming problem involved in the optimization depends on the choice of the performance index, the prediction model and constraints. For certain problem classes, the optimal RHC controller can be calculated explicitly in a piecewise affine state-feedback form [9] without the need for online optimization. However in general, the objective function and constraints might be nonlinear or the complexity of the equivalent gain-scheduled controller might warrant the use of an online optimization solver. If computational requirements necessitate and the software infrastructure enables such task to be incorporated in the real-time digital control system, online optimization based techniques offer a very powerful way to deal with constraints and changes in the controlled system. On the other hand, using these methodologies it becomes much more challenging to design the control system to meet real-time computational requirements.

In practice, control engineers invest significant effort in coding the implementation of their RHC algorithm on a particular hardware platform. This process involves addressing software engineering issues related to real-time execution requirements and process scheduling. The main purpose of

the RHC API is to provide a high level software interface that builds upon the services of the OCP to simplify implementation of receding horizon control schemes in a real-time embedded environment.

The fundamental concept of the RHC API is to create an interface at the level of mathematical programs that are involved in all online optimization based receding horizon schemes. The user is responsible for formulating the receding horizon problem at hand as a mathematical program of specific characteristics, and the RHC API provides the link to the appropriate optimization solver and real-time scheduler, which could be thought of as a plug-in module of the software. A different RHC formulation may result in a different mathematical program. However, as long as a particular formulation belongs to a certain problem class, it is the role of the RHC API to automate the code generation below the mathematical program level and interface with the required solver to perform the optimization. Besides providing this natural interface at the problem formulation level, the API makes use of OCP services which support implementing the receding horizon scheme in real-time, according to the philosophy that will be described in Section III-B.

The prototype RHC API is implemented under the OCP environment (Controls API) and uses its own problem formulation (more specific abstraction layer than a general mathematical program). It relies on LSSOL to solve the Quadratic Program (QP) that results from the problem formulation module, which takes generic RHC problem parameter inputs from the user. This specific RHC API implementation is described in the following section.

A. Generic RHC problem formulation

The RHC API version implemented as part of the DARPA SEC flight test experiment formulates and solves a quadratic optimization problem during each time frame. In this particular implementation, the interface to the user is provided at a higher abstraction layer, in the form of a generic RHC problem formulation based on linear prediction models, linear constraints and a quadratic performance index [10]. Formulating the mathematical program (QP) and invoking the optimization solver is performed automatically. The standard generic RHC problem formulation considers the following minimization problem

$$\begin{aligned}\min_v \textstyle\sum_{k=0}^{H-1} \; & x_k^T Q x_k \; + \; u_k^T R u_k \\ & + \; \left[Cx_k - Gd_k\right]^T M \left[Cx_k - Gd_k\right] \\ & + \; v^T F v \; + \; K^T v \; + \; x_H^T \Phi x_H\end{aligned} \tag{1}$$

subject to

$$a_{x,box} \leq \quad x_k \quad \leq b_{x,box} \quad {\scriptstyle k=1,\ldots,H} \tag{2a}$$

$$a_x \leq \quad L_x x_k \quad \leq b_x \quad {\scriptstyle k=1,\ldots,H} \tag{2b}$$

$$a_{u,box} \leq \quad u_k \quad \leq b_{u,box} \quad {\scriptstyle k=0,\ldots,H-1} \tag{2c}$$

$$a_u \leq \quad L_u u_k \quad \leq b_u \quad {\scriptstyle k=0,\ldots,H-1} \tag{2d}$$

$$a_{G_u} \leq \quad L_{G_u} \begin{bmatrix} x_{1:H} \\ u_{0:H-1} \end{bmatrix} \quad \leq b_{G_u} \tag{2e}$$

$$a_{G_v} \leq \quad L_{G_v} \begin{bmatrix} x_{1:H} \\ v \end{bmatrix} \quad \leq b_{G_v} \tag{2f}$$

Here, the linear dynamics of the system are specified by the matrices A, B, and E such that

$$x_{k+1} = Ax_k + Bu_x + Ed_k \quad \text{for} \quad k = 0, \ldots, H-1 \tag{3}$$

where the signal d represents both estimated disturbance and desired trajectories. The control input u sequence is determined from the optimization variables v according to

$$u_{0:H-1} := [u_0, \ldots, u_{H-1}]' = \mathcal{U} v \tag{4}$$

The columns of $\mathcal{U}$ form a basis for the control input subspace. As a result, the control input signal can then be written as linear combinations of the columns of $\mathcal{U}$.

Constraints (2a)-(2d) are targeted towards typical operational constraints. The last two constraint types (2e)-(2f) allow the user to specify general linear constraints involving states, inputs and optimization variables. The formulation allows the matrices involved in the problem to be parameter-dependent. These matrices are evaluated each time the problem formulation is called based on a parameter vector δ.

The problem formulation subroutine maps the system model and cost function matrices into the variables which parameterize the quadratic program, namely $\left(\bar{Q}, \bar{L}, \bar{Z}, \bar{a}, \bar{W}, \bar{b}\right)$. The resulting problem can then be represented as a quadratic program of the form

$$\min_v \qquad \frac{1}{2} v^T \bar{Q} v + v^T \bar{L} + \bar{Z} \tag{5a}$$

$$\text{subject to} \qquad \bar{a} \leq \bar{W} v \leq \bar{b} \tag{5b}$$

Once formulated, the parameters $\left(\bar{Q}, \bar{L}, \bar{Z}, \bar{a}, \bar{W}, \bar{b}\right)$ are passed to the quadratic program solver, LSSOL.

The method of LSSOL [11] is a two-phase (primal) quadratic programming method. The two phases of the method are: finding an initial feasible point by minimizing the sum of infeasibilities (the *feasibility phase*), and minimizing the quadratic objective function within the feasible region (the *optimality phase*). The computations in both phases are performed by the same subroutines. The two-phase nature of the algorithm is reflected by changing the function being minimized from the sum of infeasibilities to the quadratic objective function. Once any iterate is feasible, all subsequent iterates remain feasible.

LSSOL has been designed to be efficient when used to solve a *sequence* of related problems. From this aspect, it is well suited for solving receding horizon control problems or to be applied in a sequential quadratic programming method for nonlinearly constrained optimization (e.g. in the NPSOL package [12]). If the constrained problem is infeasible, LSSOL returns a flag and the RHC API uses a subroutine to automatically reformulate a relaxed problem with additional slack variables. Constraint softening is done using user-specified weights which define the relative "cost" of relaxing individual constraints [8].

Alternate instances of the generic problem formulation relying on different matrices are referred to as *modes*. Each set of data associated with a specific problem formulation has a corresponding mode number. These numbers are collected in a mode vector I that indicates which mode the entire system is in at the moment.

The RHC API requires the user to specify the following data (some in the form of mode parameter-dependent matrices) before the software component can be built:

- dimensions of quantities
- dynamics
- cost function
- constraints
- weights for constraint relaxation
- basis set for input
- warm-start function (described in Section III-B)

In general, an iterative process is required to solve a quadratic program. The following section describes the RHC API timing architecture and the term "iteration" will be used to refer to the number of iterations in the *optimality phase* and the *feasibility phase* together using LSSOL as the optimization solver.

B. Conceptual timing architecture

The RHC API is implemented as three different tasks within the OCP environment. These tasks are referred to as `Pre`, `Opt` and `Post`. Their execution spans a single major frame of the scheduler, which runs at 2 Hz for the specific application example of the DARPA SEC flight test. The minor frame rate was specified to be 20 Hz. These three tasks implement different parts of the receding horizon control scheme based on user supplied data. The RHC API formulates, solves and computes the proper control action to be taken according to the following policy.

At time step k, the RHC API component receives the following data (exogenous inputs) from other OCP components: state estimate ($x_{est}(k)$), signal estimate ($d_{est}(k : k + H - 1)$), mode vector (I), parameter vector (δ). The hard real-time task `Pre` runs at the beginning of each major frame and is responsible for executing two primary operations. It calls the user-supplied warm-start function and formulates the RHC problem based on current measurements and user-supplied data. The linearly constrained, quadratic program is constructed using the state and signal estimates, the reference signal, and the dynamics, constraint and cost matrices according to current values of I and δ.

The warm-start function has to perform two calculations. First, initialize the decision variable v, based on current inputs and data from the previous frame. It also computes a candidate baseline control action u_0 using any method that has a guaranteed completion time. Although problem-dependent, the user is expected to provide worst-case execution times for both of these steps, so they can be scheduled in hard real-time fashion.

The problem formulation is then passed to the anytime task `Opt`, which is enabled to run after `Pre` is finished. It is important to note that `Pre` is designed to have a small worst-case execution time. `Opt` solves the constrained optimization problem by calling the appropriate mathematical program solver. If the problem is infeasible, a relaxed version is solved after reformulation.

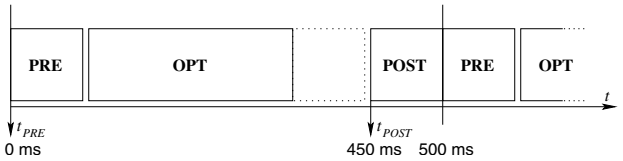

Fig. 1. Scheduling of the `Pre`, `Opt` and `Post` tasks within the RHC API.

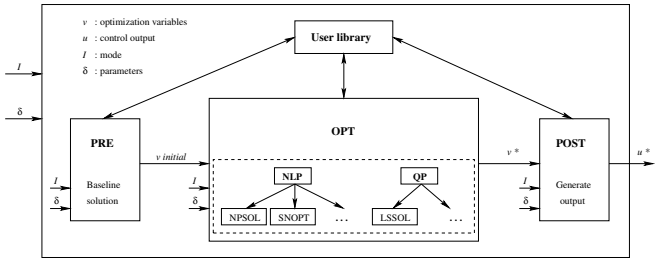

Fig. 2. Conceptual signal-flow architecture of the `Pre`, `Opt` and `Post` tasks within the RHC API.

Task Name	Task Type	Description
`Pre`	Hard Real-Time	RHC problem formulation, candidate control action.
`Opt`	Anytime	Solves constrained optimization problem, relaxation if necessary.
`Post`	Hard Real-Time	RHC problem solution processing, desired control action.

TABLE I
DESCRIPTION OF RHC API TASKS.

One minor frame before the next 2 Hz major frame, i.e. 450 ms after the start of the current major frame, `Opt` is terminated by the scheduler and `Post` is called. `Post` is a hard real-time task, which processes the most recent (either intermediate or final) optimal iterate v^* of the solver. The corresponding control action u^* is calculated and a decision is made whether to use the baseline control action u_0 or u^*. The baseline solution can be considered for implementation in case the solver could not complete a single iteration, or the intermediate solution quality is not acceptable due to infeasibility or high cost. Finally, `Post` outputs the desired control action at the end of the major frame.

The timing of the three basic tasks of the RHC API is depicted in Figure 1. Descriptions of the `Pre`, `Opt` and `Post` tasks are summarized in Table I.

The main motivation behind this scheduling scheme is to allow a variable amount of computational time for the most computationally intensive calculations. The online optimization is performed iteration by iteration in an "anytime" fashion, so it can make use of more computational time if made available by the scheduler.

Figure 2 illustrates the signal-flow relationship between the three RHC API tasks and the user libraries. The user libraries include RHC problem formulation data (parameters, prediction models and any other user-defined input data) as a collection of possible modes. The libraries should also provide the user-defined functionalities within the `Pre` and `Post` processes, such as the warm-start function and the control solution calculation.

C. Implementation details

The RHC API implementation for the DARPA SEC flight experiment was designed with the specific needs of this application example in mind. The quadratic program solver LSSOL has been modified in order to allow recovery of intermediate iterates following premature termination from other processes. User-specified code segments were implemented

using an object oriented software engineering approach to flight code design [13]. Extra care had to be taken to make sure that the user-specified code contains no dynamic memory allocations, which is a strict requirement in real-time software applications.

IV. DARPA SEC FIXED WING DEMONSTRATION EXPERIMENT

The main objective of the University of Minnesota / University of California Berkeley flight experiment was to demonstrate the use of advanced receding horizon guidance technologies, which are enabled by the real-time software tools of the RHC API and the OCP middleware.

The testbed was a full-scale T-33 jet aircraft outfitted with an autopilot and the avionics package of the X-45 UCAV unmanned aircraft. The control inputs of ground speed, turn rate and altitude were implement using an autopilot, which performed the lower level control of the aircraft. The autopilot was flight certified and safe to operate, which reduced the verification requirements on the online optimization based control law.

The basic experimental scenario of our team focused on two control objectives. First, tracking of a time-stamped position reference trajectory while respecting constraints on the vehicle dynamics. Second, detection of a simulated fault, which was artificially inserted into the system. The scenario timeline was flexible and could be influenced by the ground operator invoking pop-up threats that had to be avoided. These events resulted eventually in a reference trajectory that is depicted in Figure 3. In a second, more ambitious scenario, the fault is not removed from the system after detection, and the RHC controller is reconfigured to adapt to the faulty vehicle dynamics. Constraints are also adjusted to restrict the aircraft's maneuvers.

Description of the fault detection filter design and test results are omitted in this paper and can be found in [8], [14], along with simulations that show successful reconfiguration of the RHC controller after a fault. Flight test results of the RHC controller are also reported in [15], [16].

A. Simulation and flight test results

The receding horizon guidance controller was tested in simulation with different wind conditions and showed excellent robustness. With the exception of a few time samples, the optimization converged within two iterations throughout the entire simulation, without being terminated by the scheduler. The top two plots in Figure 3 show simulated tracking performance in the north-east coordinate frame and in terms of altitude indicating excellent tracking performance. The reference trajectory resulted from no-fly-zone and pop-up threat avoidance, as well as reaching a stationary target and performing fault detection experiments.

The bottom two plots in Figure 3 illustrate results from the flight test, which took place at Edwards Air Force Base in the Mojave desert in June 2004. The main reason for deviations from the reference trajectory and degraded tracking performance during flight test was that automatic speed control was not available on the test platform. Tracking a time-stamped position reference trajectory requires tight control of the aircraft's velocity. However, during the flight test airspeed was controlled using manual adjustments to the throttle by the pilot. Post-flight data analysis showed an average delay of 50-100 seconds in the velocity command channel, which was not modeled in the RHC controller. The implemented RHC controller was tested only up to 10-20 samples (5-10 seconds) of unmodeled additional delay compared to the prediction model and showed acceptable degradation of performance.

Due to the extra delay in the speed command channel and the resulting model mismatch, constraints became active more frequently during the flight test. This increased the computational time required by the online optimization solver. LSSOL converged to a solution in three or four iterations during a significant percentage of the total number of computational cycles as opposed to simulation results (see Table II).

Although even four LSSOL iterations could be completed on the test platform without any frame overruns or preemptive process termination by the scheduler, the effect of limiting the maximum number of iterations was investigated in simulations. The purpose of these tests was to verify the RHC API implementation and study the loss of performance as the computational time available for online optimization becomes more restricted.

An artificial delay of 50 seconds was inserted in the simulation setup, which was not modeled in the RHC predictions. The objective was to mimic the flight test environment more accurately and increase the likelihood of higher iteration numbers in the optimization thread. The resulting performance was similar to the one observed during flight tests and is depicted in the top two plots in Figure 3. Distribution of computational cycles by number of iterations is reported in Table II. Since the percentage of major frames requiring four LSSOL iterations was negligible, the controller performance was essentially unaffected by limiting the maximum number of iterations to three. However, a limit of two iterations per cycle resulted in a significant, but not devastating loss of performance as shown in the top two plots in Figure 3. The performance degradation can be attributed mainly to the poor quality speed control solution. The large artificial delay in the speed command channel resulted in velocity oscillations and frequent saturation of flight envelope limits in terms of true airspeed. Since this led to constraints being active most of the time, the number of iterations in the feasibility phase of LSSOL increased. Due to the artificially imposed iteration limit, after a feasible solution was found, it could not be improved upon by further iterations in the optimization phase. This led to poor quality speed control, which resulted in the significant lateral oscillations shown in the top left plot of Figure 3. As the aircraft was trying to match the time-stamped position reference, it veered away to "bleed off" time when travelling with higher than optimal groundspeed. These results illustrate that the RHC API was able to handle situations where the available computational time was restricted by enforcing artificial limits on the allowable number of LSSOL iterations.

ACKNOWLEDGEMENTS

This work was funded by the Defense Advanced Research Projects Agency under the Software Enabled Control program, Dr. John Bay Program Manager. The contract number

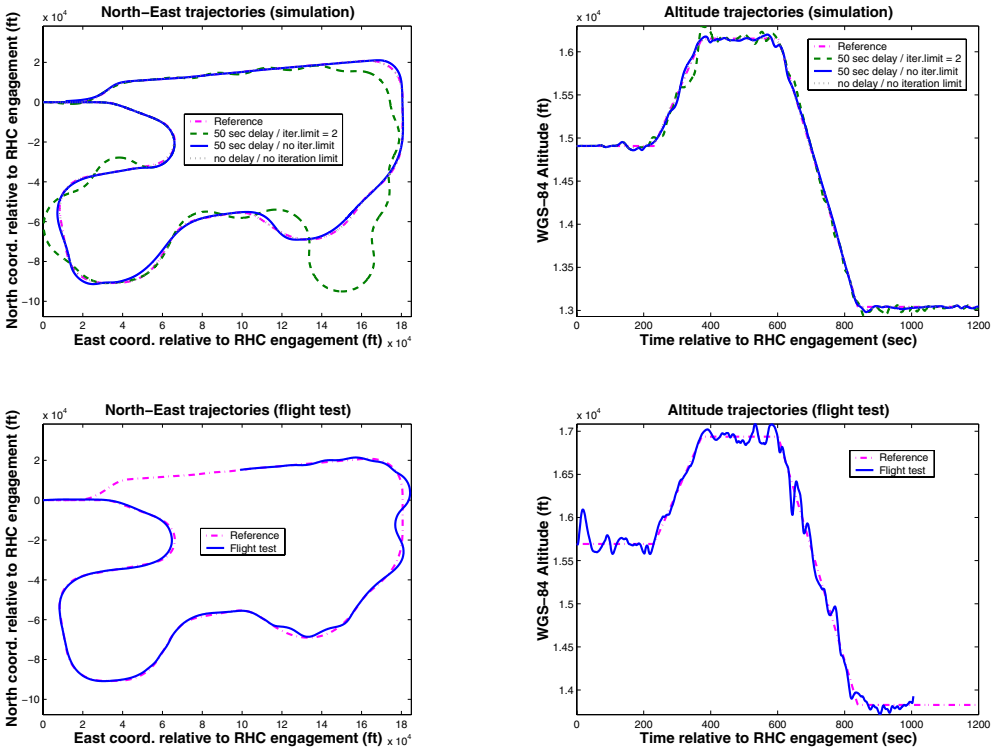

Fig. 3. North, east and altitude tracking performance. The top two plots show simulation results with artificial delay on speed command and different LSSOL iteration limits, the bottom two plots represent flight test data.

Test cases	V_{cmd} delay	iter. limit	Number of LSSOL iterations 1	2	3	4
Nominal sim.	0 s	none	0.04%	99.92%	0.04%	0.00%
Flight test	N/A	none	1.10%	59.63%	38.92%	0.35%
Simulation	50 s	none	0.04%	65.81%	33.94%	0.21%
Simulation	50 s	3	0.04%	65.85%	34.11%	0.00%
Simulation	50 s	2	0.04%	99.96%	0.00%	0.00%

TABLE II

DISTRIBUTION OF COMPUTATIONAL CYCLES BY REQUIRED NUMBER OF LSSOL ITERATIONS.

is USAF/AFMC F33615-99-C-1497, Dale Van Cleave is the Technical Contract Monitor.

The authors would like to acknowledge Brian Mendel, Tim Espey and Jared Rosson at the Boeing Company for their help and support with the integration of the RHC API into the Open Control Platform. The authors would also like to acknowledge Kenneth Hsu, Sean Estill, Zachary Jarvis-Wloszek and Raktim Bhattacharya for their work on different versions of the RHC API implementation.

REFERENCES

[1] T. Samad and G. J. Balas, *Software-Enabled Control: Information Technology for Dynamical Systems*. Wiley – IEEE Press, 2003.

[2] J. S. Bay and B. S. Heck, "Special section: Software-enabled control," *IEEE Control Systems Magazine*, vol. 23, no. 1, Feb. 2003.

[3] J. Paunicka, B. Mendel, and D. Corman, "The OCP – an open middleware solution for embedded systems," in *Proc. of the American Control Conf.*, Arlington, VA, 2001, pp. 3345–3350.

[4] J. L. Paunicka, B. R. Mendel, and D. E. Corman, *Software-Enabled Control*. Wiley – IEEE Press, 2003, ch. Open Control Platform: a software platform supporting advances in UAV control technology, pp. 38–62.

[5] Object Management Group, "Realtime CORBA joint revised submission," OMG Document orbos, Tech. Rep. 99-02-12, Mar. 1999.

[6] D. Rosu, K. Schwan, S. Yalmanchili, and R. Jha, "On adaptive resource allocation for complex real-time applications," in *Proc. IEEE Real-Time Systems Symposium*, Dec. 1997.

[7] M. Agrawal, D. Coffer, and T. Samad, "Real-time adaptive resource management for advanced avionics," *IEEE Control Systems Magazine*, vol. 23, no. 1, pp. 76–88, Feb. 2003.

[8] T. Keviczky, R. Ingvalson, H. Rotstein, A. Packard, O. R. Natale, and G. J. Balas, "An integrated multi-layer approach to software enabled control: Mission planning to vehicle control," Dept. of Aerospace Eng. and Mechanics. Univ. of Minnesota, Minneapolis. Dept. of Mechanical Eng. Univ. of California, Berkeley, Tech. Rep., Nov. 2004. [Online]. Available: http:// www.aem.umn.edu / people / faculty / balas / darpa_sec / SEC.Publications.html

[9] F. Borrelli, *Constrained Optimal Control of Linear and Hybrid Systems*, ser. Lecture Notes in Control and Information Sciences. Springer, 2003, vol. 290.

[10] S. M. Estill, "Real-time receding horizon control: Application programmer interface employing LSSOL," Dept. of Mechanical Eng. Univ. of California, Berkeley, Tech. Rep., Dec. 2003.

[11] P. E. Gill, S. J. Hammarling, W. Murray, M. A. Saunders, and M. H. Wright, *User's Guide for LSSOL (Version 1.0): a FORTRAN package for constrained linear least-squares and convex quadratic programming*, Systems Optimization Laboratory, Department of Operations Research, Stanford University, 1986.

[12] P. E. Gill, W. Murray, M. A. Saunders, and M. H. Wright, *User's Guide for NPSOL 5.0: a FORTRAN package for nonlinear programming*, University of California, San Diego. Stanford University. Bell Laboratories., 1998.

[13] O. R. Natale, "Models and tools for advanced real-time control systems," Ph.D. dissertation, Department of Engineering, Università degli Studi del Sannio in Benevento, Benevento, Italy, 2004.

[14] H. P. Rotstein, R. Ingvalson, T. Keviczky, and G. J. Balas, "Input-dependent threshold function for an actuator fault detection filter," in *16th IFAC World Congress*, Prague, Czech Republic, July 2005.

[15] T. Keviczky and G. J. Balas, "Flight test of a receding horizon controller for autonomous UAV guidance," in *Proc. of the American Control Conf.*, Portland, Oregon, June 2005.

[16] ——, "Software-enabled receding horizon control for autonomous UAV guidance," *AIAA Journal of Guidance, Control, and Dynamics*, 2005, to appear.