Washington University in St. Louis
Washington University Open Scholarship

All Computer Science and Engineering Research | Computer Science and Engineering

Report Number: wucse-2009-27

2009

The Design and Performance of Cyber-Physical Middleware for Real-Time Hybrid Structural Testing

Huang-Ming Huang, Xiuyu Gao, Terry Tidewell, and Christopher Gill

Real-time hybrid testing of civil structures, in which computational models and physical components must be integrated with high fidelity at run-time represents a grand challenge in the emerging area of cyber-physical systems. Actuator dynamics, complex interactions among computers and physical components, and computation and communication delays all must be managed carefully to achieve accurate tests. To address these challenges, we have developed a novel middleware for integrating cyber and physical components flexibly and with suitable timing behavior within a Cyber-physical Instrument for Real-time hybrid Structural Testing (CIRST). This paper makes three main contributions to the state of the art... **Read complete abstract on page 2.**

Follow this and additional works at: https://openscholarship.wustl.edu/cse_research

 Part of the Computer Engineering Commons, and the Computer Sciences Commons

Recommended Citation

Huang, Huang-Ming; Gao, Xiuyu; Tidewell, Terry; and Gill, Christopher, "The Design and Performance of Cyber-Physical Middleware for Real-Time Hybrid Structural Testing" Report Number: wucse-2009-27 (2009). *All Computer Science and Engineering Research.*
https://openscholarship.wustl.edu/cse_research/12

Department of Computer Science & Engineering - Washington University in St. Louis
Campus Box 1045 - St. Louis, MO - 63130 - ph: (314) 935-6160.

This technical report is available at Washington University Open Scholarship: https://openscholarship.wustl.edu/cse_research/12

The Design and Performance of Cyber-Physical Middleware for Real-Time Hybrid Structural Testing

Huang-Ming Huang, Xiuyu Gao, Terry Tidewell, and Christopher Gill

Complete Abstract:

Real-time hybrid testing of civil structures, in which computational models and physical components must be integrated with high fidelity at run-time represents a grand challenge in the emerging area of cyber-physical systems. Actuator dynamics, complex interactions among computers and physical components, and computation and communication delays all must be managed carefully to achieve accurate tests. To address these challenges, we have developed a novel middleware for integrating cyber and physical components flexibly and with suitable timing behavior within a Cyber-physical Instrument for Real-time hybrid Structural Testing (CIRST). This paper makes three main contributions to the state of the art in middleware for cyber-physical systems: (1) a novel middleware architecture within which cyber-physical components can be integrated flexibly through XML-based configuration specifications, (2) an efficient middleware implementation in C++ that can maintain necessary real-time performance, and (3) a case study that evaluates the middleware's performance and demonstrates its suitability for real-time hybrid testing.

Department of Computer Science & Engineering

Washington University in St. Louis
School of Engineering & Applied Science

2009-27

The Design and Performance of Cyber-Physical Middleware for Real-Time Hybrid Structural Testing

Authors: Huang-Ming Huang, Xiuyu Gao, Terry Tidwell, Christopher Gill, Chenyang Lu, Shirley Dyke

Abstract: Real-time hybrid testing of civil structures, in which computational models and physical components must be integrated with high fidelity at run-time represents a grand challenge in the emerging area of cyber-physical systems. Actuator dynamics, complex interactions among computers and physical components, and computation and communication delays all must be managed carefully to achieve accurate tests.

To address these challenges, we have developed a novel middleware for integrating cyber and physical components flexibly and with suitable timing behavior within a Cyber-physical Instrument for Real-time hybrid Structural Testing (CIRST). This paper makes three main contributions to the state of the art in middleware for cyber-physical systems: (1) a novel middleware architecture within which cyber-physical components can be integrated flexibly through XML-based configuration specifications, (2) an efficient middleware implementation in C++ that can maintain necessary real-time performance, and (3) a case study that evaluates the middleware's performance and demonstrates its suitability for real-time hybrid testing.

Type of Report: Other

The Design and Performance of Cyber-Physical Middleware for Real-Time Hybrid Structural Testing

Huang-Ming Huang[1], Xiuyu Gao[2], Terry Tidwell[1],
Christopher Gill[1], Chenyang Lu[1], Shirley Dyke[2]
[1]Department of Computer Science and Engineering
{hh1, ttidwell, cdgill, lu}@cse.wustl.edu
[2]Department Mechanical, Aerospace, and Structural Engineering
xg2@cec.wustl.edu, sdyke@wustl.edu
Washington University, St. Louis, MO, USA

***Abstract*—Real-time hybrid testing of civil structures, in which computational models and physical components must be integrated with high fidelity at run-time represents a grand challenge in the emerging area of cyber-physical systems. Actuator dynamics, complex interactions among computers and physical components, and computation and communication delays all must be managed carefully to achieve accurate tests.**

To address these challenges, we have developed a novel middleware for integrating cyber and physical components flexibly and with suitable timing behavior within a* Cyber-physical Instrument for Real-time hybrid Structural Testing (CIRST).** ***This paper makes three main contributions to the state of the art in middleware for cyber-physical systems: (1) a novel middleware architecture within which cyber-physical components can be integrated flexibly through XML-based configuration specifications, (2) an efficient middleware implementation in C++ that can maintain necessary real-time performance, and (3) a case study that evaluates the middleware's performance and demonstrates its suitability for real-time hybrid testing.

I. Introduction

Structural Engineering increasingly relies on sophisticated computational monitoring and control systems involving mechanical and structural components, and these cyber-physical systems must be tested and validated using similarly sophisticated techniques. While high-fidelity validation is critical to the acceptance of structural monitoring and control systems, in many applications testing numerous possible scenarios of a new structural control device or a new monitoring system is not feasible due to the cost and time required for such a comprehensive test. For instance, in developing civil infrastructure, the performance of a new vibration suppression system (e.g., for earthquake or hurricane mitigation) usually cannot be validated at full scale prior to its implementation (e.g., on a large bridge). In the current state of the art, the cost of such testing prohibits performing more than a few representative tests at small scales relative to the massive sizes of the structures, and thus additional validation steps are needed. Furthermore, these tests are often destructive, so that only one test can be performed with each test specimen.

This research was supported in part by NSF grants CNS-0821713 (MRI) CNS-0448554 (CAREER) and CCF-0448562 (CAREER).

As we noted in [1], the scale of civil structures often makes it infeasible to test them fully through empirical techniques alone. While reduced scale testing is useful, it cannot always capture important behaviors of the full scale structure, even if scaling effects have been carefully considered. Numerical simulation is therefore an equally important technique in modern structural analysis, and has benefited significantly by leveraging hardware innovations that offer improved computational capabilities. However, experimental validation is still essential to examine the underlying assumptions made by the numerical models, especially considering the existence of highly nonlinear elements under extreme dynamic loading.

Hybrid testing, which integrates both physical components of the structure of interest and computational models of other known structural components, thus improves significantly on either purely numerical or purely empirical approaches. Due to the lack of real-time support for hybrid testing, however, hybrid testing at a slow (a.k.a. *pseudodynamic*) time scale is the state of the art [2], [3], [4], [5]. Unfortunately, testing at such time scales may not reveal critical dynamic system features, necessitating a real-time approach. The leap from slow time scales to real-time raises significant research challenges such as real-time coordination, fault tolerance and control stability. These issues in turn require the development and use of new kinds of cyber-physical instruments for real-time hybrid testing.

In our previous work [1] we developed an initial prototype of a *Cyber-physical Instrument for Real-time hybrid Structural Testing* (CIRST), to illustrate the feasibility of our vision and to gain insights into the design and implementation challenges for developing robust reusable middleware for such systems. In this paper we present three main contributions to the state of the art in middleware for cyber-physical systems: (1) a novel middleware architecture for CIRST within which cyber-physical components can be integrated flexibly through XML-based configura-

tion specifications, (2) an efficient implementation of that architecture in C++ that can maintain necessary real-time performance, and (3) a case study that evaluates the CIRST middleware's performance and demonstrates its suitability for real-time hybrid testing.

Section II surveys related work in the areas of automatic configuration and hybrid testing. In Section III we describe the cyber-physical middleware design and implementation requirements imposed by real-time hybrid testing. Section IV presents the design and implementation of the CIRST middleware to meet those requirements. In Section V we present a case study that applies the CIRST middleware to different real-time hybrid testing scenarios and evaluates its performance. We summarize the contributions of this research and describe future work in Section VI.

II. Related Work

In addition to a real-time *target system* connected to hydraulic actuators, inner loop controllers, sensors, analog-to-digital (A/D) and digital-to-analog (D/A) converters, a separate host computer is often used for hybrid testing system design, and for monitoring and visualization of test results. Host systems do not require real-time capabilities and are generally realized by standard desktop computers. However, the target system is the center of a real-time hybrid testing system and its design has the greatest impact on the accuracy of the entire system and thus on the fidelity of the experiments run with it. In this section we therefore focus primarily on prior research into the computational aspects of target systems for hybrid testing and consider only a representative sample of other relevant software tools.

Target Systems: A controlled system can be specified using high level Simulink models and then compiled into object code for target systems. The object code is then linked with a light weight real-time kernel which provides basic interrupts and I/O services to generate executables. dSPACE's TargetLink [6] is an integrated toolset which includes the Simulink model compiler and a real-time kernel to produce executables in their hardware platforms. The xPC target [7] from Mathworks is a similar toolset that targets generic x86 hardware instead. For example, the platform developed at UIUC [8] targets more generic x86 hardware, allowing greater flexibility in the experimental equipment. The major benefit of both solutions is that the tools provide streamlined environments from model definition and evaluation to target deployment that requires little programming for basic systems.

However, for more complicated hybrid testing systems with hundreds of degrees of freedom, more complex non-linear material and structural models are needed, which are core elements of the OpenSees [9] open source structural analysis framework adopted by NEES [10]. OpenSees also uses object oriented programming to provide tools for re-usably specifying numerical models for simulations. OpenSees is purely for computation of the response of the numerical model, and provides no built-in support for real-time operation. NEES runs it on Phar Lap ETS (an real-time OS which provides a subset of Win32 APIs to minimize the effort for porting desktop application to embedded systems) in order to achieve real-time performance.

Model Driven Development: The MARTE[11] UML profile specification adds capabilities for model-driven development of real-time and embedded systems, including specification, design, verification, and validation support to improve communication between developers and interoperability between tools from different vendors. Our research is currently focused on more fundamental middleware issues, though code generation and verification from MARTE models may be beneficial as future work.

III. Middleware Requirements

In this section we describe the requirements imposed by real-time hybrid testing, which motivate and guide the design and implementation of the CIRST middleware presented in Section IV and a case study of the CIRST middleware as a platform for hybrid testing presented in Section V. These requirements fall into three main categories: encapsulation of concerns, specialized infrastructure, and flexible configuration, which we now consider in detail.

Encapsulation of Concerns: The first requirement imposed by real-time hybrid testing systems is that the middleware provide a mechanism for encapsulation of different cyber and physical abstractions, so that different elements of a real-time hybrid testing system can be added or removed without inordinate impact on the other components of the system. For example, a small scale experiment may use a physical actuator and a physical test specimen to obtain models of their behaviors, and then a larger scale experiment may integrate different instances of those models with the physical test specimen and actuator. Encapsulation of each actual or simulated element allows it to be re-used and configured independent of the others.

This kind of encapsulation is the norm in existing hybrid testing platforms such as dSPACE [6] (which leverages Simulink's intuitive block based design to allow users to decompose systems into constituent parts) and OpenSees (which offers an object oriented approach for specifying experiments, thereby promoting reusability and the ability to better manage the complexity inherent in complicated experiment designs). Any cyber-physical middleware for real-time hybrid testing must therefore offer comparable encapsulation capabilities.

Specialized Infrastructure: While the previously mentioned requirement has led to the design of reusable middleware frameworks (e.g., CIAO [12], TAO [13], nORB [14] and ACE [15]) to constrain system complexity while maintaining real-time performance for other application domains, the nuances of real-time hybrid testing require further innovation in middleware design and implementation beyond those frameworks.

The scale and complexity of experiments that already can be conducted using existing hybrid testing approaches such as dSPACE [6], OpenSees [9], and NEES [10] imposes a stingent performance requirement for middleware that would support similarly rich hybrid testing in real-time. To achieve suitable real-time performance, data and event flows through the system must be flexible, correct, efficient, and temporally predictable even at small time scales (e.g., to support hybrid testing of a multi-story civil structure at 1280Hz).

Implementation complexity is an important concern for the kinds of fine-grained time scales that must be achieved, and based on our previous middleware design experiences [14], specialization of the supporting middleware infrastructure to enforce correctness and temporal predictability (e.g., low level data handling to avoid unnecessary copying, or routing data along multiple paths) is the second crucial requirement.

Flexible Configuration: The third requirement imposed by real-time hybrid testing systems is that configuration capabilities must allow system developers to specify *both declaratively and flexibly* which elements will constitute the system, and how those elements will be initialized and interconnected. Such configurability allows both recurring questions and new ones to be investigated through the assembly and deployment of real-time hybrid structural testing systems from common sets of reusable elements..

In addition to easing the task of specifying system configurations, these capabilities also must reduce the risk of configuration errors. Our previous experience developing a prototype system for conducting simple hybrid testing experiments [1] often involved tedious and error-prone experimental configuration changes, which motivates the importance of this requirement. Without suitable tools for system wide configuration of cross-cutting cyber and physical concerns (e.g., to avoid type errors due to inconsistent representation of physical units as data moves through the system), implementing those experimental configuration changes required code changes throughout the prototype system, which in turn risked introducing further problems that would need to be addressed.

IV. Design and Implementation

To address the requirements discussed in Section III, the CIRST middleware is designed to: (1) *encapsulate* system abstractions as components, (2) provide *specialized infrastructure* to enforce type safety and timing properties within and between components, and (3) provide *flexible configuration* capabilities end-to-end.

A. Encapsulation

The CIRST middleware allows computational functionality to be encapsulated as distinct *components* with defined input and output interfaces through which to communicate with other components. Unlike traditional object oriented frameworks, there is no notion of an inheritance hierarchy between components in the CIRST middleware, since real-time hybrid testing is largely data-flow oriented.

Communication between components is based on a set of interfaces, which we refer to as *flow ports* since different data types can be configured to flow between components through them. Flow ports that receive data from other components are called *in-ports*; and those that send data to other components are called *out-ports*. A component `comp1` can deliver data to another component `comp2` when a connection is made between an out-port of `comp1` to an in-port of `comp2`. In the CIRST middleware, a connection can be established via a single C++ statement in the following style:

```
comp1.out_port() = comp2.in_port();
```

Given two components a and b and a connection between an out-port of a and an in-port of b, we refer to a as the *upstream component* and b as the *downstream* component for that connection.

B. Specialized Infrastructure

The CIRST middleware infrastructure has four important areas of specialization: component ports, data movement, multicast support, and timing constraint handling. Component ports enforce enforce type safety by checking compatibility along a connection of the data type sent by an out-port and the data type expected by an in-port. The CIRST middleware infrastructure avoids data unnecessary copying by allowing an upstream component to declare whether the data it sends should be modified by its downstream components, and allowing downstream components to decide whether to copy its input data or reuse the input buffer. The CIRST middleware infrastructure provides a multicast capability by allowing the out-port of a component to be connected to multiple in-ports of other components and preserves the integrity of the data passed between the connections such that all downstream components see the same data value passed from their in-ports. Finally, the CIRST middleware infrastructure allows timing constraints to be associated with a component, and appropriate handlers to dispatched if a constraint is violated at run-time.

Component Ports: The CIRST middleware infrastructure uses a polymorphic function wrapper (`std::tr1::function`) [16] and a generic callback mechanism to implement out-ports. To simplify the management of flow ports we set the arity of the function wrapper to 1 and the return type as int (for indicating error conditions); therefore, a typical out-port which sends a non-modifiable data of type `T` would be declared as `std::tr1::function<int (const T&)> out_port();`. In-ports, on the other hand, are implemented as reference wrappers (`std::tr1::reference_wrapper`) or function objects which allows an overloaded `operator()` to set up a connection with an out-port as shown previously in this section.

Data is passed between components using a `FlowData` class template with two type parameters, `ContentType` and `TagType`. To ensure type safety, the CIRST middleware infrastructure uses concrete data types instead void* or abstract data types to pass data between components. The `ContentType` parameter gives the actual data type to be processed by a component (e.g., a vector or matrix of numeric data for a numerical simulation computation). The `TagType` is used for data synchronization between different data flows. If the test structure is sufficiently complex, the computation time of a required finite element method may be longer than the period between sensor inputs or actuator outputs, in which case the numeric computation must be divided into several parts and dispatched to different physical processors or CPU/GPU cores for parallel computation. The `TagType` then is used to combine the results of different parallel sub-computations. Other uses of the `TagType` include allowing filtering out data (e.g., if it arrives with higher frequency but has lesser importance to a particular experiment).

Data Movement: To avoid unnecessary copying, we simulate the r-value reference (or *move semantics*) in C++0x [17] with a special `Moveable` template to pass flow data from an upstream component to a downstream component when the data passed is no longer needed by the upstream component. This allows a downstream component to optimize by reusing existing memory and avoid unnecessary copying. A `Moveable<T>` object contains a pointer to `T` which can be accessed via `operator->`; in addition, the `Moveable` template provides an type conversion operator to convert an object of `Moveable<T>` to `const T&`. For run-time efficiency, a function invocation with statement `fun(Moveable(obj));` can be dispatched correctly to `void fun(Moveable<T>& obj)` when available or to `int fun(const T& obj)` if the moveable version does not exist.

An alternative would have been to use a simple non-const reference to play the role of `Moveable` in above example. However, it is not obvious which version would be used by the the caller for a statement like `fun(obj)`. Furthermore, under the current C++ standard, `std::tr1::function` is not able to distinguish correctly between two overload functions with const and non-const references. That is, given an object declared as `std::tr1::function<int (const T&)> callback;` and a function declared as `void fun(T&)`, the statement `callback=fun` can be compiled without any warning even though the caller expects the callee not to modify its parameter, yet the compiler allows the callee to change its input argument. Using `Moveable<T>` avoids this problem, and therefore the CIRST middleware infrastructure uses `Moveable<T>` instead of `T&` to indicate that the data passed from a port can be modified directly by a downstream component.

With the the introduction of move semantics for component interfaces, a component in-port is implemented by overloading `operator()` : one of them takes a constant reference parameter, and the other takes a `Moveable` parameter. Thus if the out-port of a component is of type `std::tr1::function<int(const T&)>` it can only bind to the member function with constant reference parameter; if the out-port is of type is of type `std::tr1::function<int(Moveable<T>)>`, it can bind to either the constant reference or the moveable parameter member function, depending on the availability of the moveable parameter member function in the in-port's interface.

Multicast Support: The CIRST middleware is specialized to support multicast from the out-port of a component to the in-ports of several downstream components. This is implemented in the CIRST middleware via an intermediate `FlowSplitter` component to connect the upstream component and downstream components. If the data passed by the upstream component is of type `Moveable<T>`, the `FlowSplitter` will invoke the member function with constant reference parameter unless it is the last downstream component of the given out-port to receive that data. The `boost::signal`[18] capability provides a similar multicast functionality, but does not offer the optimization to pass moveable data to the last downstream component. We can achieve this optimization only because we fixed the arity of each in-port and out-port to be 1, while `boost::signal` allows arbitrary arity.

Timing Constraint Handling: The CIRST middleware is specialized to allow each component to be optionally enabled with timing constraint checking. The required frequency and acceptable arrival jitter in the invocations of an in-port of a component can be specified. We also support three different policies for how to handle deadline misses: *skip*, *continue* or *stop*. In the *skip* policy, the computational operation of the component simply won't

be invoked and the out-ports of the component will not be called; however, the experiment can still go on. With the *continue* policy, not only will the experiment keep executing but, the component will still be invoked and its outputs produced as though no deadline had been missed. With the *stop* policy, an exception will be thrown when a deadline is missed, to stop the experiment and perform required cleanup operations. With any of these three policies, if the data arrived *earlier* than expected, the component's computation operation is simply delayed until its expected start time.

C. Flexible Configuration

To ease configuration through declarative specifications, and to make CIRST accessible to users without C++ background, the CIRST middleware provides XML based configuration scripting capability to set up and run a cyber-physical experiment. XML configuration file specifies the components used by an experiment, the connections between components, the configuration parameters needed by each component, the timing constraints for each component, and the timing instrumentation points to be used for performance analysis. We achieve this capability through a code generator that reads the XML configuration file and emits C++ code that can be compiled into an executable program. The configuration file can also be read by the executable program when it starts up, to change the component configuration parameters without regenerating and re-compiling the C++ code. However, it is not possible to configure other aspects of the system without regenerate the C++ code in our current implementation.

The CIRST middleware provides further support for flexible configuration by allowing component port adapters to be generated from the XML configuration file. If components out-ports and in-ports cannot be connected directly due to data type mismatches, but an acceptable data conversion exists (e.g., between different units of measure) the CIRST middleware allows users to specify data conversion expressions that are parsed by the code generator and used to generate C++ components that perform the specified conversion. By relieving system developers of the tedious and error-prone responsibility to either ensure exact type matching among component ports or to code C++ components to perform the necessary conversions, the automatic generation of these adapters further assists system developers in using the CIRST middleware. flexibly, correctly, and easily.

V. Case Study

In this section we present a case study that (1) describes a representative experimental setup for real-time hybrid testing, (2) shows how the CIRST middleware can be applied to develop and run different cyber-physical experiments using that experimental setup, and (3) evaluates the performance of the middleware and of the resulting real-time hybrid testing system as a whole. The results of those evaluations confirm the efficacy of the CIRST middleware for running accurate real-time hybrid tests and comparing the performance of alternate configurations of physical and computational components in those tests.

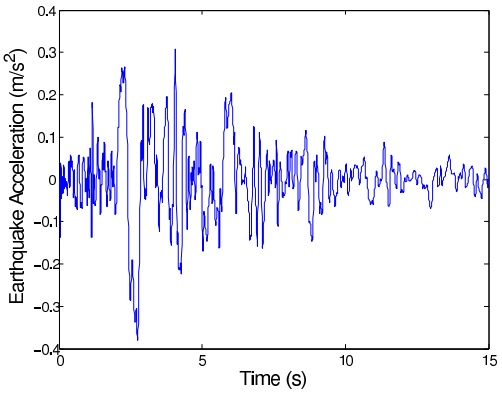

Figure 1. Input ground motion

A. Experimental Setup

The *physical test specimen* used in our experimental setup is composed of a small scale steel compression spring which is used to represent the bending stiffness of an actual column in a portal frame structure. The linear elastic spring has a nominal stiffness of 37.6 kN/m (215 lb/in) and a maximum allowable deflection of 7 cm (2.77 in).

The rest of frame are all modeled in a *computational substructure* that assumes (1) each column has same stiffness value as the physical test specimen, and (2) classical proportional damping of 2% for each structure mode. Earthquake acceleration data taken from Woodward-clyde Federal Services database, and shown in Figure 1, are used as a ground motion input to excite the computational subsstructure. The data is scaled down in magnitude to avoid a situation in which the maximum structure response might exceed the deflection limit of the experimental setup.

A Shore-Western 910D double-ended hydraulic *actuator* is employed as the loading device to drive the physical test specimen. The actuator has a maximum stroke of 6 inches, with a built-in concentric linear variable differential transformer (LVDT) for ready integration into a position feedback control system. A Schenck-Pegasus 162M servovalve rated for 15 GPM at a 1,000 psi pressure drop is used to control the actuator. The servo-valve has a nominal operational frequency range of 0-60 Hz and is driven by a Schenck-Pegasus 5910 digital controller. An Omega load cell with a range of 1 kip is included in series with the physical test specimen to measure the restoring force.

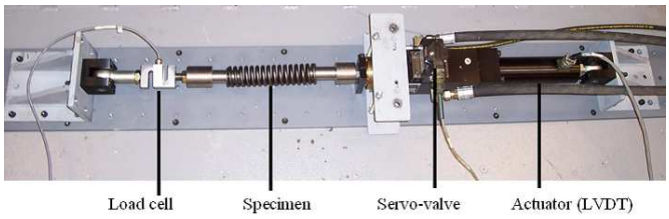

Figure 2. Experimental setup for hydraulic structure testing

K_p	3	mA/in	controller proportional gain
τ_v	2.76e-3	s	servo-valve time constant
K_q	26.959	$in^3/s/mA$	valve flow gain
K_c	2.499e-4	$in^3/s/psi$	valve flow pressure gain
A	0.652	in^2	piston area
C_l	2.476e-5	$in^3/s/psi$	piston leakage coefficient
V_t	36.59	in^3	volume of fluid
β_e	96153	psi	effective bulk modulus
m_t	1.754e-2	$lb-s^2/in$	mass of test specimen
c_t	8.429	$lb-s/in$	viscous damping coefficient
k	212.62	lb/in	stiffness of specimen

Table I
IDENTIFIED ACTUATOR PARAMETERS

The experimental setup is shown in Figure 2. Experiments conducted using this setup were performed in the Washington University Structural Control and Earthquake Engineering Laboratory, which houses a hydraulic pump that can be operated at 3,000 psi with maximum flow rate of 43 GPM.

B. System Model

An experimental transfer function for the overall physical component was obtained under a band-limited white noise excitation signal with a bandwidth of 50 Hz and magnitude root mean square of 0.07 cm (0.028 inch). Actuator parameters are identified using a nonlinear least square optimization routine to curve-fit the experimental data, values for each parameter as well as its physical meaning are shown in Table I.

As can be seen in Figure 3, the curve-fitted model represents the actuator dynamics well within a 0-50 Hz frequency range. For this reason we use it for the controller design to compensate actuator dynamics. Since stability and accuracy are the major concerns for dynamic analysis, the applied hybrid testing methodology is studied by comparing a prototype single degree of freedom (SDOF) hybrid system with a reference system whose input (excitation force) and output (displacement) relationship is governed by transfer function Equation 1:

$$G_{anyl} = \frac{1}{Ms^2 + Cs + K} \tag{1}$$

The closed-loop hybrid system is shown schematically in Figure 4 and can be described by the transfer function

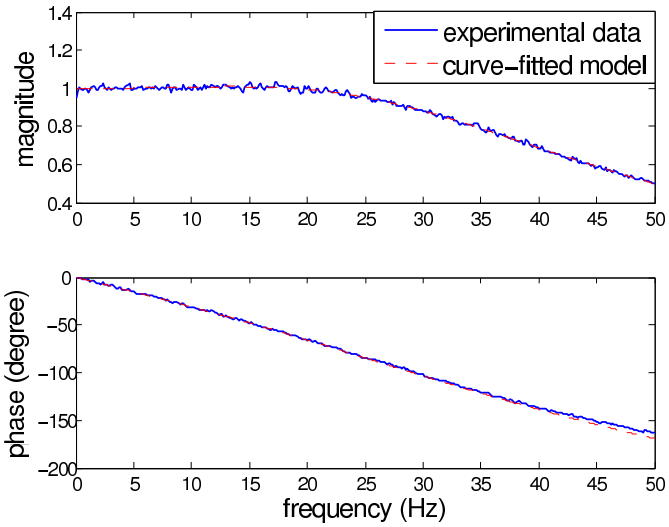

Figure 3. Experiment transfer function versus model.

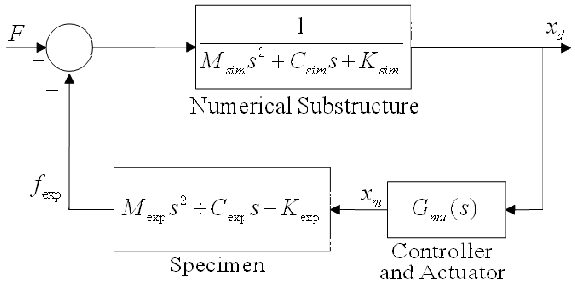

Figure 4. Closed-loop hybrid system

shown in Equation 2. Equation 3 represents experimental components' transfer functions between the input command and output displacement measurements.

$$G_{hyd} = \frac{1}{M_{sim}s^2 + C_{sim}s + K_{sim} + G_{mu}(s) \cdot [M_{exp}s^2 + C_{exp}s + K_{exp}]} \tag{2}$$

$$G_{mu}(s) = \frac{K_p \frac{K_q A}{K_c}}{H(s)} \tag{3}$$

where $H(s) =$

$$(\tfrac{V_t}{4\beta_e K_c} m_t \tau_v)s^4 + (\tfrac{V_t}{4\beta_e K_c} m_t + m_t \tau_v + \tfrac{V_t}{4\beta_e K_c} c_t \tau_v)s^3 +$$

$$(m_t + \tfrac{V_t}{4\beta_e K_c} c_t + \tfrac{A^2}{K_c}\tau_v + c_t \tau_v + \tfrac{V_t}{4\beta_e K_c} k \tau_v)s^2 +$$

$$(c_t + \tfrac{V_t}{4\beta_e K_c} k + \tfrac{A^2}{K_c} + k\tau_v)s + k + K_p \tfrac{K_q A}{K_c}.$$

Two complex conjugate poles of the hybrid system represent structure poles that are modified in the complex plane due to the introducing of servo-hydraulic system. These poles dominate the dynamic characteristics of the hybrid system, as opposed to the other four poles with

	Natural Frequency (Hz)			
Sys-1	1	2	5	10
Sys-2	1	2	5.0001	10.002
Sys-3	.9999	1.9996	4.993	9.9469

Table II
NATURAL FREQUENCIES WHEN $K_{exp} = K_{sim} = K$

	Damping			
Sys-1	0.02	0.02	0.02	0.02
Sys-2	0.0177	0.0153	0.0088	0.0007
Sys-3	0.0083	-0.0034	-0.0374	-0.0851

Table III
DAMPING RATIOS WHEN $K_{exp} = K_{sim} = K$

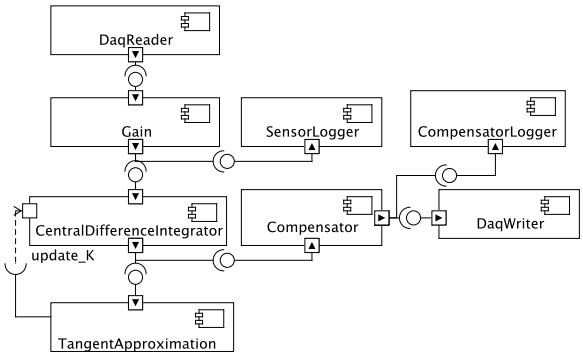

Figure 5. Component assembly

much faster dynamics. Assuming that the total mass, total damping and half of the stiffness are modeled in the computational substructure, Tables II and III show the comparison of natural frequencies and damping ratios respectively, between the reference system (sys-1) and the hybrid systems with (sys-2) and without (sys-3) compensation.

Although the natural frequency variation is small, significant damping reduction has been observed in the hybrid system. This can be improved by the use of the compensation scheme. More damping reduction is associated with the test when the natural frequency of the reference system increases, which indicates a larger error introduced by the hybrid testing methodology. Instability will occur if the damping becomes negative.

Consider also another (worst) case with zero stiffness but again with the total mass and total damping modeled in the computational substructure. Table IV and V show the natural frequencies and damping ratios of this worst case test scenario, with faster damping reduction which indicated degraded stability and less accuracy.

Through this simple example, we can conclude that overall hybrid system behavior depends not only on local behaviors such as accurate modeling and good compensation strategy, but also intrinsically on the experimental plan and setup, e.g. on the proper partition of physical properties between the numerical and experimental elements of the system.

	Natural Frequency (Hz)			
Sys-1	1	2	5	10
Sys-2	1	2	5.0003	10.006
Sys-3	.9999	1.9991	4.9875	9.9534

Table IV
NATURAL FREQUENCIES WHEN $K_{exp} = K$ AND $K_{sim} = 0$

	Damping			
Sys-1	0.02	0.02	0.02	0.02
Sys-2	0.0153	0.0107	-0.0024	-0.0185
Sys-3	-0.0035	-0.0268	-0.0947	-0.1898

Table V
DAMPING RATIOS WHEN $K_{exp} = K$ AND $K_{sim} = 0$

C. Middleware Configuration

To demonstrate how the CIRST middleware can be applied to develop and run different cyber-physical experiments using the experimental setup described in Section V-A and the system model described in Section V-B, we configured a component assembly that integrates a commercially available I/O device and C++ and Matlab simulation components on a standard Linux machine.

The CIRST middleware component assembly, implemented by C++ code generated from an XML configuration file, is shown in Figure 5. We ran the component assembly on a Dual Pentium 4 Xeon 2.40 GHz processor machine with one gigabyte of RAM. For communication with the analog sensors and actuators in the experimental setup, we used a National Instruments Data Acquisition Board (BNC-2120 DAB). The `DaqReader` and `DaqWriter` components in Figure 5 represent the software components for reading from and writing to the data acquisition board.

Each simulation step executes the required computation following the order of flow ports in the assembly and then generates commands to the actuators. Our initial assembly used the onboard clock from the data acquisition board to trigger each simulation step. However, due to unforseen variation in the triggering times produced by that mechanism we replaced it with a software based triggering mechanism that gave much more regular timing behavior, an experience that further illustrates the utility of a configurable component middleware approach to real-time hybrid testing.

D. Experimental Evaluation

To evaluate the scalability of our middleware, we first extend our experiment to use a sequence of n-story structures (for different values of n) each of which is equivalent

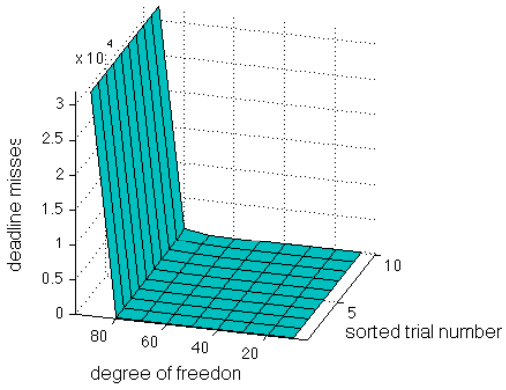

Figure 6. The deadline misses of a n-story structure experiment

in our system model to an n degree-of-freedom (DOF) system; however, only one column of the first story is captured from the physical rigid beam specimen, and data for all other stories are purely based on analytical computation. We used a 15 second earthquake wave and a simulation frequency of 1280Hz. We varied the value of n and for each n ran 10 trials. To assess the system's ability to perform at the specified simulation frequency we recorded the number of times that a deadline was missed (i.e, when the completion of a component's execution was past the end of the current period at that simulation frequency). The results of these trials, with the trials for each number of degrees of freedom sorted by the number of deadline misses, are shown in Figure 6. As those results show, the real-time hybrid test system had very few deadline misses as n goes up to 80. After that, however the computation load becomes too large for the system to maintain at 1280 Hz. However, even when n is small, we still see a few deadline misses in some 15 second run of the system.

Attribution of Deadline Misses: To examine the sources of these deadline misses, we instrumented the system to measure the time required for the following operations: reading sensor data, writing actuator commands and performing other numerical computations. Figure 7 shows the cumulative distributions for each of the operations individually and in aggregate, when n is 5, 60 and 90.

The results in Figure 7 show that reading from and writing to the DAQ board dominate the aggregate timing when n is 5 or 60. Table VI shows the worst case scenario for those segments. In the case of $n = 60$, the average total execution time is 554 μs , which is far below our deadline 780 μs. However, the worst case is 941 μs, which happens when the DAQ write operation spends 578 μs.

These results identify variation in the timing of the read

	5	60	90
read	401	483	520
write	439	578	627
computation	39	216	647
total	701	941	1745

Table VI
WORST CASE TIME IN μsec FOR EACH SEGMENT OF A SIMULATION STEP

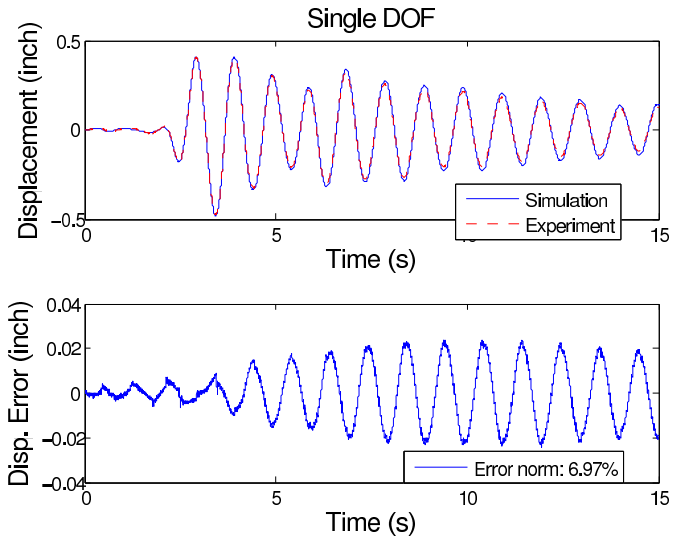

Figure 8. The hybrid test and simulation results of single DOF setup

and write DAQ board operations as an important cause of deadline misses when the computational load on the system is otherwise sufficiently low. However, since the timing of these operations is subject to the behavior of a proprietary device driver whose implementation we do not control we were left with two alternatives: (1) replace the standard off-the-shelf DAQ board we had available, or (2) evaluate the extent to which the deadline misses were affecting the fidelity of our test system, and determine whether or not the result was acceptable (or could be ameliorated if not).

Fidelity Comparison between Hybrid Tests and Simulations: We pursued the latter alternative, by comparing the fidelity of our hybrid test results obtained in the presence of those deadline misses, to comparable tests using pure simulation models that did not suffer that effect. Figures 8, 9 and 10 show the results of hybrid and simulation tests with 1, 5, and 60 degrees of freedom respectively. Each DOF indicates one lumped mass. The single DOF test assumes a 1910 kg floor mass and the other two tests assume a 175 kg mass per structure floor.

The error norm in each case is calculated as the percentage ratio between the standard deviation of the displacement error and the simulated displacement. As shown in the Figures, the error norm decreases as the DOF increases, even though the 60 DOFs case introduces more deadline

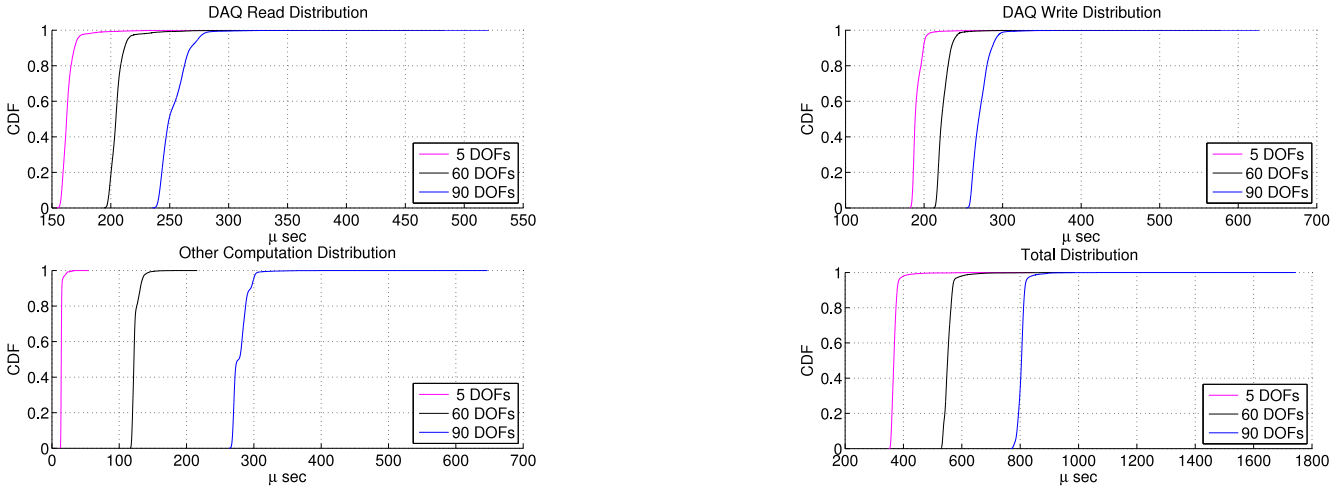

Figure 7. The cost of DAQ read/write and other computations

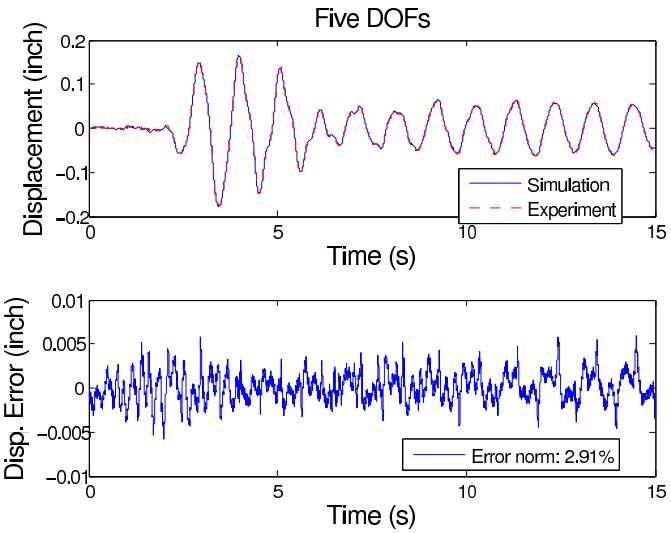

Figure 9. The hybrid test and simulation results of 5 DOFs setup

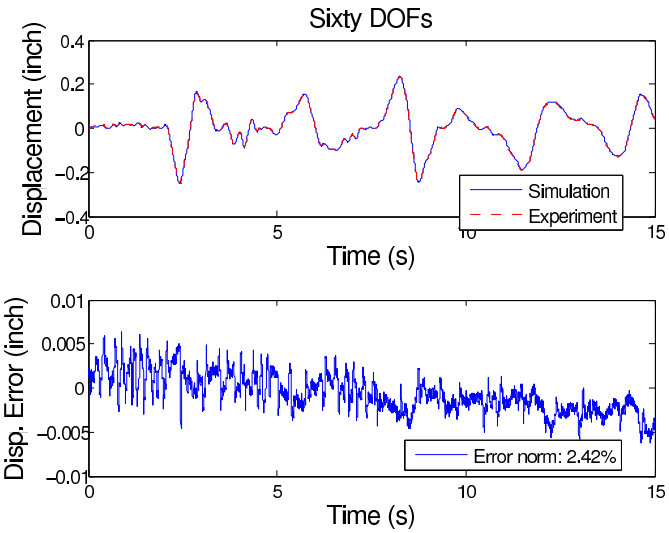

Figure 10. The hybrid test and simulation results of 60 DOFs setup

misses. This is because the test errors are mostly associated with physical component. In our experimental setup, only a single column of the bottom story are actually represented by the physical components and all others are simulation model. As the DOF increases, the errors introduced by the physical components in the system will have less effect on the fidelity of system. Therefore, the possible errors introduced by any source would have less impact on the results with heavier computational loads than with lighter ones in this case.

The data obtained from the real-time hybrid experiments matched well with the simulation results for all 3 tests, which indicates that the infrequent deadline misses did not have a large impact on the fidelity of the hybrid experiments. However, since the error between the hybrid tests and the simulations could be attributed to a variety of possible sources including imperfect sensor measurement, modeling inaccuracy which does not consider nonlinearity of the physical test specimen, or insufficiently detailed timing information about the physical specimen or the actuator, further study is needed to fully quantify the effects of deadline misses in the cyber portions of our real-time hybrid testing system.

VI. Conclusions and Future Work

In this paper we have described how cyber-physical system domains like real-time hybrid motivate development of novel middleware infrastructures that can address the kinds of requirements described in Section III. In Section IV we presented the design and implementation of a novel mid-

dleware infrastructure for the Cyber-physical Instrument for Real-time hybrid Structural Testing (CIRST) we are developing in this research. The case study in Section V demonstrates both the suitability of the CIRST middleware for real-time hybrid testing, and the need for further investigation into remaining sources of timing variability and other factors that may impact real-time hybrid test systems.

Our immediate future work will also focus on further modeling and evaluation of the DAQ read and write timing variability, and on potential mitigation strategies that may be needed for particular real-time hybrid testing scenarios that are sensitive to those variations. We plan to expand the use of parallel execution of the numeric simulation algorithms, aided by the data synchronization features of the CIRST middleware, to achieve and quantify further scalability of our approach through aggregation of more computational resources.

We plan to improve our hydraulic actuator model to capture the dynamics of physical components over a broader frequency range, as well as to capture associated nonlinearity. In conjunction with study the effects of jitter in when the command displacement signal is sent to the actuator, we plan to explore other dynamic compensation strategies in order to apply the desired signal to test specimen more accurately.

REFERENCES

[1] T. Tidwell, X. Gao, H.-M. Huang, C. Lu, S. Dyke, and C. Gill, "Towards Configurable Real-Time Hybrid Structural Testing: A Cyber-Physical Systems Approach," in 12^{th} *International Symposium on Object-Oriented Real-time Distributed Computing (ISORC)*, (Tokyo, Japan), IEEE, Mar. 2009.

[2] M. Ahmadizadeh, G. Mosqueda, and A. Reinhorn, "Compensation of actuator delay and dynamics for real-time hybrid structural simulation," *Earthquake Engineering and Structural Dynamics*, vol. 37, no. 11, pp. 21–42, 2008.

[3] J. Carrion and S. B.F., "A model based delay compensation approach for real time hybrid testing," in 4^{th} *International Conference on Earthquake Engineering*, (Taipei,Taiwan), 2006.

[4] S. A. Mahin and P. B. Shing, "Pseudodynamic method for seismic testing," *Journal of Structural Engineering*, vol. 111, no. 7, pp. 1482–1503, 1985.

[5] S. Mahin, P. Shing, C. Thewalt, and R. Hanson, "Pseudodynamic test method. Current status and future directions," *Journal of Structural Engineering*, vol. 115, no. 8, pp. 2113–2128, 1989.

[6] "dSPACE systems." http://www.dspace.de/ww/en/inc/home.cfm.

[7] "xPC target 4.1, perform real-time rapid prototyping and hardware-in-the-loop simulation using PC hardware." http://www.mathworks.com/products/xpctarget/.

[8] J. Carrion and B. F. Spencer, "Model-based strategies for real-time hybrid testing," Tech. Rep. NSEL-006, University of Illinois at Urbana-Champaign, 2007.

[9] "OpenSees, a software framework for developing applications to simulate the performance of structural and geotechnical systems subjected to earthquakes." http://opensees.berkeley.edu/.

[10] P. B. Shing, Z. Wei, R. Y. Jung, and E. Stauffer, "NEES fast hybrid test system at the University of Colorado," in 13^{th} *World Conference on Earthquake Engineering*, (Vancouver, Canada), 2004.

[11] "The object management group. 2005. uml profile for modeling and analysis of real-time and embedded systems. omg request for proposals, real-time/05-02-06.." http://www.omg.org/cgi-bin/doc?real-time/05-02-06.pdf.

[12] V. Subramonian, L.-J. Shen, C. Gill, and N. Wang, "The design and performance of configurable component middleware for distributed real-time and embedded systems," in *RTSS '04: Proceedings of the 25th IEEE International Real-Time Systems Symposium*, (Washington, DC, USA), pp. 252–261, IEEE Computer Society, 2004.

[13] Institute for Software Integrated Systems, "The ACE ORB (TAO)." www.dre.vanderbilt.edu/TAO/, Vanderbilt University.

[14] V. Subramonian, G. Xing, C. Gill, C. Lu, and R. Cytron, "Middleware specialization for memory-constrained networked embedded systems," in *Proceedings of 10th IEEE Real-time and Embedded Technology and Applications Symposium (RTAS)*, 2004.

[15] D. C. Schmidt, "ACE: an Object-Oriented Framework for Developing Distributed Applications," in *Proceedings of the* 6^{th} *USENIX C++ Technical Conference*, (Cambridge, Massachusetts), USENIX Association, Apr. 1994.

[16] "Jtc1/sc22/wg21. draft technical report on c++ library extensions." http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1836.pdf, 2005-06-24.

[17] "Working draft for the c++ language." http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2800.pdf, 2009-3-23.

[18] "Boost c++ library." http://www.boost.org.