6th International Conference on Advances in Experimental Structural Engineering
11th International Workshop on Advanced Smart Materials and Smart Structures Technology
August 1-2, 2015, University of Illinois, Urbana-Champaign, United States

A Uniform Method to Integrate Testing Equipment for Large-Scale Quasi-Static Structural Testing

K.J. Wang[1], K.C. Tsai[2]

1 *Technologist,National Center for Research on Earthquake Engineering, Taipei, Taiwan.*
E-mail: kjwang@narlabs.org.tw
2 *Professor, Dept. of Civil Engineering, National Taiwan University, Taipei, Taiwan.*
E-mail: kctsai@ntu.edu.tw

ABSTRACT
This paper presents a method that allows smooth integration of different testing equipment such as actuator controllers and data acquisition systems for the purpose of conducting large-scale quasi-static structural testing. Rather than using less equipment each of which has high testing capacity, a more economical and hence more practical approach is instead to jointly use more equipment each of which provides only moderate testing capacity. This research work designed and established a uniform method that achieves this goal by means of software integration without losing the flexibility that can be offered in quasi-static structural tests.
A new application protocol, ñRemote Equipment Control and Data Exchange (Recdex),ò was proposed for the communication need and was implemented in newly developed C++ classes in the ñSoftware Framework for Quasi-static Structural Testing (SFQSST)ò earlier developed by National Center for Research on Earthquake Engineering, Taiwan to utilize the Internet to achieve the said goal. A cyclic test of 1/13-scale model of RC nuclear containment vessel was conducted using the proposed method. In this test, twelve actuators, four actuator controllers, and one data acquisition system were controlled by four host computers. The success of the test demonstrated the validness of the proposed method.

KEYWORDS: *quasi-static, Internet, large-scale, C++, SFQSST*

1. INTRODUCTION

Undoubtedly structural testing plays an irreplaceable role in the advancement of the development of structural knowledge. Structural testing is required not only to verify the validness of newly developed theories, but also to evaluate the properties of structural components, assemblages, as well as systems. In some studies the method of structural testing is also used to evaluate the seismic properties of existing structures. In recent years, more and more testing requests have been placed on laboratories. Researchers increasingly request testing on specimens not only with more delicate details but also with larger sizes.

Among all structural testing methods the quasi-static testing has the longest history of application in structural laboratories. The definition of quasi-static testing is that the rate of specimen deformation is not the main control target in tests. That is, when conducting quasi-static tests typically only the displacement or the force levels are considered as the control targets, without paying much attention to the rates at which the displacement or the force commands are applied. This characteristic of not controlling the deformation rates lends the testing method to perfectly appropriately test specimens made of materials whose responses are practically independent of their deformation rates, such as steel and concrete, the most widely used materials in civil engineering. Since the deformation rates are insignificant, typically quasi-static tests are conducted by repeating a series of ñexecution steps.ò The concept of an execution step is illustrated in Fig. 1.1. In each execution step, the actuators are firstly controlled to ramp to their target levels and then controlled to hold the specimen still at the achieved levels for a certain amount of time. The purpose of introducing the pause stage after the ramp stage is to perform certain tasks including data acquisition, immediate data manipulation, calculation of command levels corresponding to next execution step, and the human activity of investigation on the specimen.

Compared to shake table testing, in quasi-static testing the controls are not required to cope with the inertial and damping force introduced during the course of the tests. In other words, the size of the specimen that can be tested in quasi-static testing is significantly larger. This eliminates the problems that are frequently encountered in shake table testing, such as scale effects and specimen-table interaction. In most cases tests upon large-scale or full-scale specimens can only be planned and executed by quasi-static testing method.

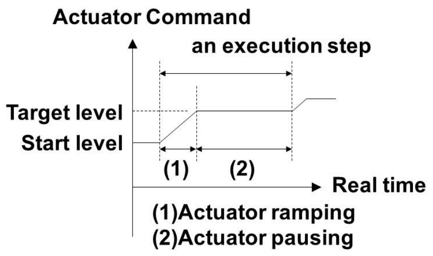


Figure 1.1 Definition of an execution step

In addition, in quasi-static testing the tests need not be executed in hard real-time. This results to two favorable features: (1) testing devices with only moderate working rates can be accepted, and (2) it only needs to consider the logic, but not the timing as in real-time systems, in the interaction and communication between different used apparatuses. The first feature actually translates to the characteristic of cost effectiveness which implies that more testing equipment can be purchased within budget. The second feature suggests that the testing system can be one with a more open architecture. It is generally easier to add or replace testing hardware and/or software components to add or update the testing capacities of a quasi-static testing system.

The characteristics described above make the quasi-static testing more suitable to test specimens with larger sizes and with more delicate modelling details. However, to perform a test on such specimen usually it requires a large number of testing apparatuses, such hydraulic actuators and data acquisition channels. One possible way for the laboratory to provide such high testing capacity is to use high-capacity hardware such as a single actuator controller that can simultaneously control the motion of a large number of hydraulic actuators. However, it might not be the most cost effective investment to equip the laboratory with such high-capacity hardware since after all such demand of testing on large or delicate specimens is still less than that for those tests that need only decent or moderate testing capacity. Contrarily, it would be more feasible to link a large number of moderate-capacity hardware together by means of software programs to collaboratively provide the required higher testing capacity. This research developed the required software components that integrate experimental equipment in a uniform manner for large-scale quasi-static testing.

2. A FLEXIBLE SOFTWARE FRAMEWORK FOR QUASI-STATIC TESTING

An object-oriented C++ software framework, named ñSoftware Framework for Quasi-Static Structural Testing (SFQSST)ò, that solves the most frequently encountered problems in quasi-static testing was developed in National Center for Research on Earthquake Engineering (NCREE), Taipei, Taiwan. Some of the flexibility SFQSST provides to the end users of quasi-static testing include:

(1) New functionality provided by new testing hardware can be easily added to the existing testing system.
(2) New functionality provided by new procedures of data manipulation can be easily added to the existing testing system.
(3) There is a universal method to synchronize all participating hardware and software components. Communication between any two hardware machines need not depend on the built-in hardware input/output capacity.
(4) Users need not write new or modify existing program codes to support new actuator configurations. Instead, users can define new actuator configurations by editing the input files.
(5) Tasks to be performed in an execution step are not hardcoded in the control program and can also be defined by editing input files. They can even be modified dynamically during the course of the test.
(6) Commands can be added, deleted, and modified during the course of the test.
(7) New signals of virtual sensors can be defined dynamically during the course of the tests. Users supply mathematical formulae to define those virtual signals. For example, the user can create a signal that calculates the axial force of a brace element based on the strain gauge readings measured on that brace.
(8) Data collected by different hardware machines and software components can be simultaneously compared or further collectively manipulated during the course of the test.

Detailed description of SFQSST can be found in [1]. Three important designs in SFQSST that relate to support of integration of testing equipment are described in this section.

2.1. Machine and Trigger in Quasi-static Testing

The abstraction **Machine** encapsulates the concept of an entity (a software component or hardware machine) that can be request to do specific works. For example, a concrete **Machine** object can be a command generator, an actuator controller, a data acquisition system, a data file writer, a data uploader (to a remote database management system), or a software calculatoré etc. Some of the **Machine** classes are shown in Fig. 2.1. Each concrete **Machine** object performs certain task which is a part of the complete tasks in an execution step. The user defines each **Machine** object such that the **Machine** object knows its specific task in detail when it is requested to do work at a later time. However, they do not have the knowledge as to when they should perform the specified tasks. Each individual task (such as actuator ramping, data acquisition, virtual signal calculation, data saving in files, data uploading to a remote databaseé , etc.) that should be performed in a quasi-static test is encapsulated in a concrete **Machine** object. This design allows easy modification on existing tasks and easy addition of new tasks since each task is completed encapsulated in its own concrete **Machine** class.

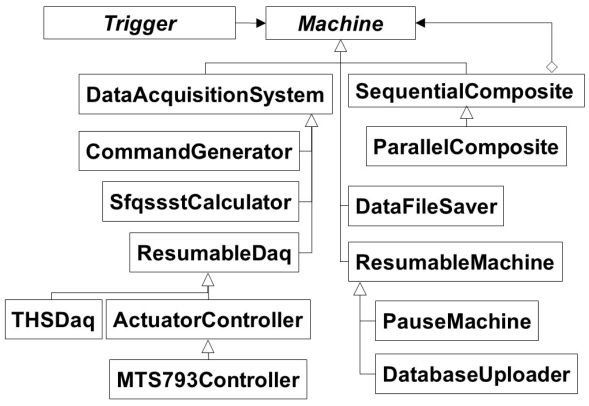
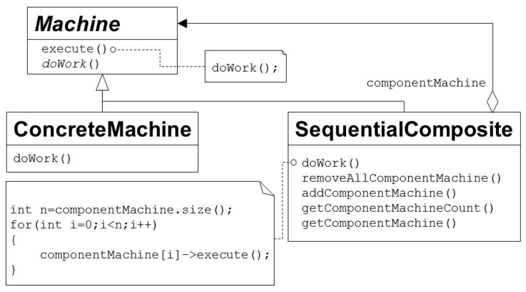


Figure 2.1 The Trigger class and Machine class hierarchy

The abstraction **Trigger** is the window for the user to make requests to a concrete **Machine** object to perform its specified tasks. A concrete **Trigger** object creates a working thread in which it issues work requests to a specified concrete **Machine** object. The user uses a concrete **Trigger** object to start, pause, resume, and stop the sequence of a test.

2.2. Task Contents in an Execution Step

One of the most important features that quasi-static testing can offer is flexibility. In quasi-static testing the tasks to be performed in an execution step vary from one test to another depending on each individual test goal. In addition, in real practice there even exist unexpected situations that require dynamic change of the tasks to be performed in an execution step during the course of the test. SFQSST adopts the Composite Pattern [2], as shown in Fig. 2.2, to provide such flexibility. The end user can change the task contents in an execution step by inserting or deleting various concrete **Machine** objects into a **SequentialComposite** or a **ParallelComposite** object (both are composite machines), which receives the triggering request from the concrete **Trigger** object. By appropriately arranging the component **Machine** objects in a composite machine object, task contents can be changed according to each test goal without the need to modify existing and develop new program codes. The task content can even be changed dynamically during the course of the test according to various unexpected test conditions (such as the requirement to change the loading protocol or to add/delete data measuring channels due to unexpected specimen damages).



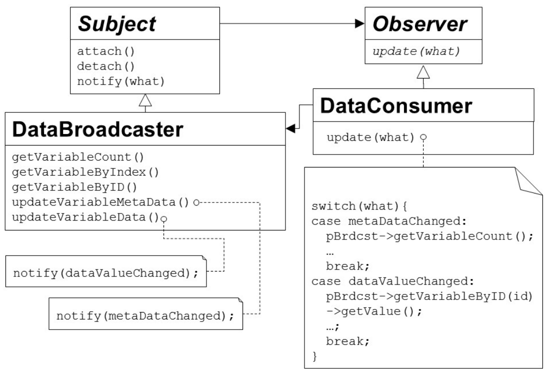
```
int n=componentMachine.size();
for(int i=0;i<n;i++)
{
    componentMachine[i]->execute();
}
```

Figure 2.2 Composite Pattern adopted in SFQSST for flexible definition of contents in an execution step

2.3. Data Exchange between Machines and Other Observer Objects

As explained in previous sections, in SFQSST corresponding testing tasks are performed by various kind of concrete **Machine** objects. Meanwhile SFQSST provides **SequentialComposite** and **ParallelComposite** objects to allow the end users to freely compose the task contents in an execution step by arranging concrete **Machine** objects into a composite machine. The concrete Trigger object would issue work request to the composite machine and the composite machine would in turn relay the work request to its component **Machine** objects in the sequence specified by the end users. However, for some of these concrete **Machine** objects to work they might need information generated by other objects in the system. For example, for an actuator controller to control the actuator piston, the controller need to know the target level, which obviously is generated by some command generator object in the system.

SFQSST adopts the Observer Pattern [2], as shown in Fig. 2.3, to achieve the goal of data exchange between all system objects. SFQSST provides several concrete **Variable** classes to allow all system objects to store data values. **DataBroadcaster** objects are containers of **Variable** objects and is a kind of **Subject** objects. In this design, when a **DataAcquisitionSystem** object generates data in an execution step, it saves the data values to concrete **Variable** objects. The **DataBroadcaster** would then inform those **DataConsumer** objects that are interested in the corresponding data values. Thereby data values can be transmitted from one object to another. The **Variable** objects in a sense are the elements that glue all those originally independent objects together to become a workable system.



```
switch(what){
case metaDataChanged:
  pBrdcst->getVariableCount();
  …
  break;
case dataValueChanged:
  pBrdcst->getVariableByID(id)
  ->getValue();
  …;
  break;
}
```

Figure 2.3 Observer Pattern adopted in SFQSST for data exchange

This section describes the mechanisms by which SFQSST provides flexibility for quasi-static testing. It is obvious that in SFQSST multiple hardware machines (even from different vendors, have different working purposes and logics) and software components can be integrated and simultaneously utilized by one control program in a smooth manner. However, there are some cases that difficulties arise to control all the hardware machines in a single control program. Such situations might be caused by the fact that the testing machines are located in different places such that it is difficult to connect the machines to the control computer simultaneously. Another situation might be the limitation posed by the configurations of specific testing hardware that it only allows certain setup of the hosting control computer. This suggests that the concept of geographically distributed structural testing might be a solution in such scenarios to achieve further integration of testing equipment for a single quasi-static test.

3. SUPPORT FOR GEOGRAPHICALLY DISTRIBUTED QUASI-STATIC TESTING

In the past decade, several platforms for conducting geographically distributed hybrid simulation have been developed around the world [3-5]. This research establishes a new testing platform that links testing hardware and software components that are geographically distributed together to conduct a single quasi-static test. This platform is established using Transmission and Control Protocol/Internet Protocol (TCP/IP). A binary application communication protocol, named ñRemote Equipment Control and Data Exchange (Recdex),ò is proposed. Currently, only four Recdex packets are defined as shown in Table 3.1.

Table 3.1 Currently defined Recdex packets

Packet name	Usage
RECDEX_SENDSIGSET	This packet is to be sent by both the server and the client before the test starts. It contains the names of the signals that are to be sent to the remote side. When sent by the client the signal values are actually the required command values (input parameters) for the server program. When sent by the server the signal values are actually the feedback values (output responses) for the client program.
RECDEX_EXECUTE	The client program requests service from the remote server program by sending this packet. This packet carries the required command values (input parameters) for the server program. The number and the sequence of the values should correspond to those of the signal names previously sent to the server program using the RECDEX_SENDSIGSET packet.
RECDEX_EXECUTE_DONE	Once the server program receives a RECDEX_EXECUTE packet, it should perform any required action (e.g., actuator control or data acquisitioné). After it finishes the task it should send a RECDEX_EXECUTE_DONE packet back to the client program as an acknowledgement for the previously received corresponding RECDEX_EXECUTE packet. This packet carries the feedback values (output responses) for the client program. The number and the sequence of the values should corresponds to those of the signal names previously sent to the client program using the RECDEX_SENDSIGSET packet.
RECDEX_TEST_FINISHED	This packet is sent by the client program to the server program to indicate that no further request of testing service will be issued.

The architecture of the Recdex platform is illustrated in Fig. 3.1. It is obvious that this architecture is probably the simplest one since there is only one Recdex server and one Recdex client. They are connected via typical Ethernet connection and they communicate by the proposed protocol Recdex. The Recdex server provides certain testing service, such as actuator motion control, data acquisitioné etc. Recdex client requests the remote service from the Recdex server. To support Recdex in SFQSST, two main classes (**RecdexServerImp** and **RecdexClientImp**) that both implement Recdex, and one concrete **Trigger** subclass (**NetTrigger**), are designed and implemented. Partial class hierarchies of these classes are shown in Fig. 3.2 and Fig. 3.3.

As described in section 2.1, the abstraction **Trigger** is the window from which end user makes a work request upon a concrete **Machine** object. For the server program, in the scenario of geographically distributed testing, the work request actually comes from the remote client program in each execution step. The end user of the server program simply indicates whether or not the server is ready to serve. Therefore, a concrete **Trigger** class, **NetTrigger**, is designed and implemented. A **NetTrigger** object allows the end user to start the working thread. The **NetTrigger** object also issues work request to a user-specified **Machine** object (depending on what the service the server program is planned to provide) when a RECDEX_EXECUTE comes. A **NetTrigger** object is created by and associated with a **RecdexServerImp** object.

The responsibilities of **RecdexServerImp** include: (1) implementing the communication protocol Recdex, and (2) serving as the bridge between the server program itself and the remote software entity (the client program) which issues a work request in each execution step. To fulfill the second responsibility described here, the **RecdexServerImp** is designed to be publicly inherited from **DataAcquisitionSystem**. Before the test starts once it receives a list of names of input parameters carried by the RECDEX_SENDSIGSET packet, it creates a concrete **Variable** object for each of the input parameter signal. During the course of the test when in each execution it receives the input parameter values via the RECDEX_EXECUTE packet, it acts as a

DataAcquisitionSystem object to broadcast those values so that all other system objects in the server program have access to them.

The **RecdexServerImp** class is also publicly inherited from **LoginServerProtocolImp**, which in turn is inherited from the abstraction **ProtocolImplementor** which provides services of establishing TCP/IP connections, as well as sending and receiving data packets. **LoginServerProtocolImp** (along with **LoginClientProtocolImp**) provides a simple login procedure to enhance the safety of network communication.

The main responsibility of **RecdexClientImp** is to serve as an agent to remote testing service for the client program. Namely, from the view point of the client program, a **RecdexClientImp** object is a kind of concrete **Machine** object since a **Machine** is a kind object that performs certain tasks in an execution step. Therefore a **RecdexClientImp** object can be included in a **SequentialComposite** or a **ParallelComposite** object, which can be triggered by a concrete **Trigger** object in an execution step. When the **RecdexClientImp** object is requested to perform its tasks, it firstly sends a RECDEX_EXECUTE packet to the server program and then waits for the corresponding RECDEX_EXECUTE_DONE packet to come back. If the RECDEX_EXECUTE_DONE packet carries the remote execution result (the output responses), the **RecdexClientImp** stores these results in concrete **Variable** objects which were previously created when the client program received the RECDEX_SENDSIGSET packet. In this way all the other system objects in the client program can have access to the execution results sent by the remote server program.

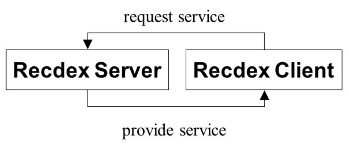
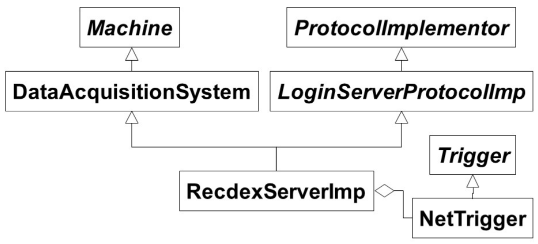


Figure 3.1 Architecture of the Recdex platform



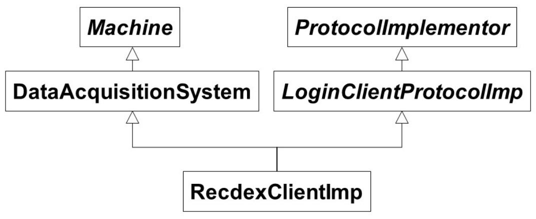


Figure 3.2 Partial inheritance of RecdexServerImp　　Figure 3.3 Partial inheritance of RecdexClientImp

An example of SDOF pseudo-dynamic testing, as shown in Fig. 3.4, is given here to illustrate how the aforementioned objects work together to achieve geographically distributed quasi-static testing. In the client program a **SequentialComposite** machine which includes two component machines, ñPseudodynamicCmdò (a **CommandGenerator**) and ñMTSControllerAgentò (a **RecdexClientImp**), is defined. In the server program a ñMTSControllerServiceò (a **RecdexServerImp**) and a ñMTSControllerò (a **MTS793Controller**) are created. In each execution step, the ñPseudodynamicCmdò calculates the displacement command value and saves it in a concrete **Variable** class (**Double**) ñDispò (step 1), such that when the ñMTSControllerAgentò is requested to work the ñMTSControllerAgentò knows where to retrieve the value of the input parameter, the displacement command (step 2). However, ñMTSControllerAgentò is merely a software entity which does not know how to impose the displacement command on the specimen. Instead, it knows how to relay the work request to a remote server program by sending the command value with a RECDEX_EXECUTE packet (step 3). Meanwhile, in the server program, the user has started a **NetTrigger** object upon a concrete **Machine** object ñMTSController,ò whose task is to impose the displacement command on the specimen. However, the **NetTrigger** object does not issue work request on ñMTSControllerò unless the ñMTSControllerServiceò allows it to. ñMTSControllerServiceò allows the said trigger to occur only when the value of the input parameter (displacement command) carried by the RECEDEX_EXECUTE packet is received and saved in a **Double** object ñDò (step 4). After that, the **NetTrigger** issues the work request on the ñMTSControllerò to impose the displacement command on the specimen. Then ñMTSControllerò retrieves its command from the **Double** object ñDò (step 5), imposes the displacement command on the specimen, and finally measures and saves the restoring force in another **Double** object ñRFò (step 6). Once this is done the **NetTrigger** knows that the current execution step is completed, then it notifies the **ProtocolImplementor** object (in this case, the **RecdexServerImp** object ñMTSControllerServiceò, which created the **NetTrigger** object in an earlier time) of this event. Then the ñMTSControllerServiceò retrieves the force value from the **Double** object ñRFò (step 7) and sends this value back to the remote client program with a RECDEX_EXECUTE_DONE packet (step 8). In the client program, the ñMTSControllerAgentò receives the force value and saves it in the **Double** object ñForceò (step 9). This completes the current execution step on the **SequentialComposite** object in the client program. When next execution step starts, the abovementioned procedure repeats with the ñPseudodynamicCmdò firstly retrieving the value of the restoring force from the **Double** object ñForceò (step 10).

In this example, before the test (the first execution step) runs, TCP connection between the client program and the server program has to be established first. ñMTSControllerAgentò and ñMTSControllerServiceò need to send RECDEX_SENDSIGSET packets to each other. The sending signal set that ñMTSControllerAgentò sends to ñMTSControllerServiceò contains only one signal, the ñDisp.ò Similarly, the sending signal set that ñMTSControllerServiceò sends to ñMTSControllerAgentò contains only one signal, the ñRF.ò

For comparison, Fig. 3.5 illustrates the details of conducting a SDOF pseudo-dynamic test in a single control program. The **SequentialComposite** object in the control program contains two component **Machine** objects, the ñPseudodynamicCmdò and the ñMTSController.ò This example explains the method proposed by this study: the uniform method to smoothly integrate testing equipment (whether or not the equipment is geographically distributed) for quasi-static testing.

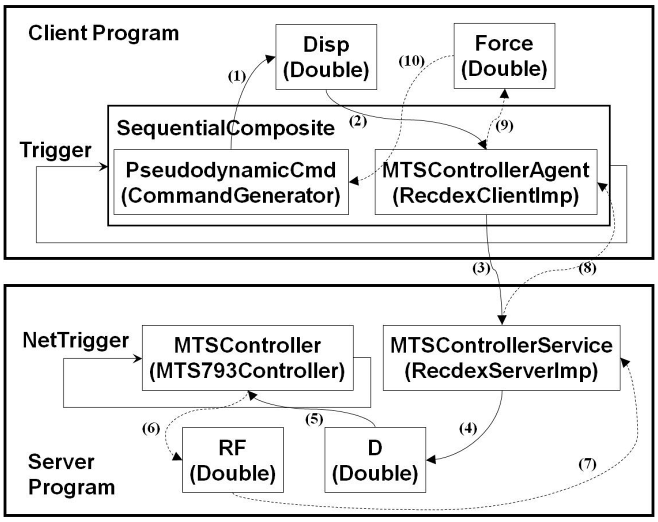


Figure 3.4 An example of geographically distributed pseudo-dynamic testing

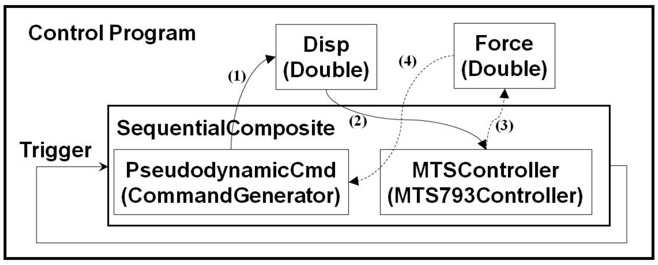


Figure 3.5 Normal implementation of pseudo-dynamic testing

4. A TEST ON A SEPCIMEN OF REINFORCED CONCRETE CONTAINMENT VESSEL

A reinforced concrete containment vessel (RCCV) is considered to be one of the key contributors to a nuclear power plant (NPP) systemôs Defense in Depth (DID) strategy. NCREE and the University of Houston, USA, cooperated to design and build a 1/13-scaled RCCV shell specimen to investigate the mechanical behavior and the failure mechanism of reinforced concrete cylindrical shells. The prototype cylindrical shell has an outer diameter of 33m, a wall thickness of 2m, and a clear height of 29.5m. This resulted in a shell specimen design of 250 cm in outer diameter, 15 cm in thickness, and 225 cm in clear height. Fig. 4.1 shows the cyclic loading test setup and lateral support system.

Totally twelve 980kN actuators were controlled by four MTS actuator controllers in this test. Eight actuators were employed in the lateral direction to apply cyclic loads. Four actuators were used in vertical direction to maintain double curvature and constant axial load of the specimen. Experimental data was acquired using TML THS1100 static data logger and a 3-D optical displacement measurement system NDI. Three instances of a control program ñFlexControlò which was developed using SFQSST were used to control the eight lateral actuators. The three instances of ñFlexControl,ò one client and two server programs, were connected using typical Ethernet connection. These three control programs communicated through Recdex and collaboratively imposed the specified cyclic displacement commands on the RCCV specimen. In each execution step, according to the desired loading protocol, the client program calculated the displacement command value and controlled the motion of two lateral actuators. Besides that, the client program also sent the command value to the other two server programs, which controlled the motion of the remaining six lateral actuators. The client program also took the responsibility of controlling the THS1100 data acquisition system. A typical test result is shown in Fig. 4.2. The success of this test demonstrated the feasibility of the method to uniformly integrate different testing equipment to collaboratively conduct quasi-static tests proposed in this study.

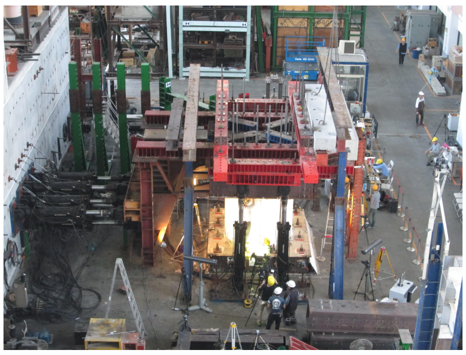


Figure 4.1 Test setup of the RCCV cyclic test

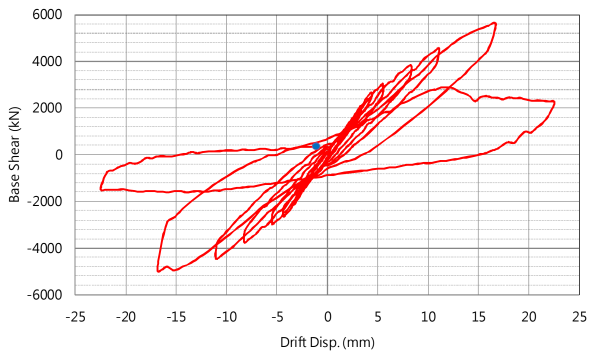


Figure 4.2 Experimentally obtained hysteresis of the RCCV specimen

5. CONCLUSION

This paper describes a uniform method that integrates different testing equipment for quasi-static testing. An application protocol, Recdex, was proposed and related C++ classes were developed in SFQSST to support geographically distributed testing. A cyclic test performed on a 1/13 scaled RCCV specimen was conducted using the method proposed by this study. The success of the test validate the feasibility of the proposed method.

AKCNOWLEDGEMENT

The authors would like to express their sincerest gratitude toward the NCREE laboratory support. Thanks also go to Dr. C. L. Wu and Dr. H. C. Yang for their enormous support in preparing the RCCV cyclic test.

REFERENCES

1. Wang, K. J. and Tsai K. C., (2011). A Software Framework for Quasi-static Structural Testing. *NCREE technical report*. NCREE-11-007.
2. Gamma, E., Helm, R., Johnson, R., Vlissides, and J. M., (1994), Design Patterns: Elements of Reusable Object-oriented Software. Addison-Wesley
3. Watanabe, E., Kitada, T., Kunitomo, S., and Nagata, K. (2001). Parallel pseudo-dynamic seismic loading test on elevated bridge system through the Internet. Proc., 8th East Asia-Pacific Conf. on Structural Engineering and Construction, Singapore.
4. Tsai, K. C., Yeh, C. C., Yang, Y. S., Wang, K. J., Wang, S. J., and Chen, P.C. (2003). Seismic hazard mitigation: Internet-based hybrid testing framework and examples. Proc., Int. Colloquium on Natural Hazard Mitigation: Methods and Applications, Villefranche sur-Mer, France.
5. Mosqueda, G. (2003). Continuous hybrid simulation with geographically distributed substructures. PhD dissertation, Dept. of Civil and Environmental Engineering, Univ. of California at Berkeley, Berkeley.