HCI^2 Workbench: A Development Tool for Multimodal Human-Computer Interaction Systems

Jie Shen, Wenzhe Shi
Department of Computing
Imperial College London
London, U.K.
js1907/ws207@doc.ic.ac.uk

Maja Pantic
Imperial College London, U.K.
EEMCS, Univ. Twente, N.L.
maja@doc.ic.ac.uk

Abstract—**In this paper, we present a novel software tool designed and implemented to simplify the development process of Multimodal Human-Computer Interaction (MHCI) systems. This tool, which is called the HCI^2 Workbench, exploits a Publish / Subscribe (P/S) architecture [13] [14] to facilitate efficient and reliable inter-module data communication and runtime system management. In addition, through a combination of SDK, software tools, and standardized description / configuration file semantics, the HCI^2 Workbench provides an easy-to-follow procedure for developing highly flexible and reusable modules. Moreover, the HCI^2 Workbench features system persistence and portability by using standardized module packaging method and system configuration files. Last but not least, usability was another major concern. Unlike other similar tool, including Psyclone [9] and ActiveMQ [10], the HCI^2 Workbench provides a complete graphical environment to support every step in a typical MHCI system development process, including module program development and debugging, module packaging, module management, system configuration, module and system testing, in a convenient and intuitive manner. To help demonstrating the HCI^2 Workbench, we also present a readily-applicable system developed using our tool. This open-source demo system, which is called the CamGame, consists of an interactive system allowing users to play a computer game using hand-held marker(s) and low-cost camera(s) instead of keyboard and mouse.**

Keywords-Development Tool; Software Framework; Multimodal Human-Computer Interface; Publish / Subscribe Architecture; Module Reusability; Centralized System Managemen; Software Usability

I. INTRODUCTION

Along with the rapid increase in computational power and network bandwidth during the past decades, the trend in the computing industry has shifted from PC-centered applications to services delivered through ubiquitous computing in a more human centered manner [1]. With this recent development, multimodal human-computer interface (MHCI) has become an emerging research topic among the community. Unlike traditional human-computer interfaces (e.g. based on keyboard, mouse, and so on), MHCI interacts with users through natural modalities including facial expression, body gesture, verbal or non-verbal vocal cues, and so on [4] [7]. Arguably, MHCI not only simplifies the application of computer systems, but also reduces user distraction and increases user satisfaction and productivity, hence has great potential in future pervasive systems [1] [2].

Nevertheless, developing MHCI is not an easy task. The difficulty comes from two aspects. Firstly, human behavior understanding and multimodal dialogue modeling are hard AI problems [1] – [6]. Secondly, because MHCI systems are normally constructed from a large number of highly interdependent and interwoven heterogeneous algorithmic units, the system integration procedure is often rather cumbersome. Among the two problems, the first one has been well perceived and investigated with great efforts, while the second one was largely overlooked by the community [12]. More specifically, most published works used custom methods for system integration, which often resulted in application-specific and non-extensible systems. While there were also some general software frameworks being used, none of them has attracted a sizable user group due to their specific limitations [12].

A. Related Works

Thorisson et al. argued that AI systems (of which MHCI systems are certainly a subset) should be built in a modular and incremental way due to their inherent complexity [14]. As a high-level guideline for software frameworks aimed at AI applications, they proposed the Constructionist Design Methodology (CDM), where the key idea is to adopt the Publish / Subscribe (P/S) architecture [7] [9] [12] [14]. In particular, CDM recommends the explicit use of channels, serving as both means for information exchange and module 'connectors' [14]. Psyclone [9] and ActiveMQ [10] are two notable software frameworks complying with CDM.

In addition to the P/S architecture, Shen and Pantic also argued that another domain-specific feature of MHCI systems considers the underlying software framework which should facilitate efficient and low-latency transmission for high bit-rate message streams [12]. This is the criterion which Psyclone and ActiveMQ fail to meet. Hence, a new software framework to tackle this problem by exploiting a shared memory based data transport protocol has been proposed in [12].

While [14] and [12] emphasized on system structure flexibility and data transmission efficiency, module reusability and the overall usability of the software framework are also

This work has been supported by the European Research Council under the ERC Starting Grant agreement no. ERC-2007-StG-203143 (MAHNOB).

important criteria. Nevertheless, all of the aforementioned software frameworks did not demonstrate sufficient consideration of these two issues. Although the P/S architecture naturally eliminated the dependency between modules [12], the modules were still dependent on their (usually hard-coded) local environment (i.e., the channels they subscribe and / or publish to), hence prohibited reusing to some extent. Regarding usability, none of the aforementioned tools featured a compact (visual) representation of the system structure. Moreover, they did not provide users with an easy mean to control the system at runtime. Lack of centralize control made the system configuration / adjustment / redistribution process counter-intuitive and hence hard to follow.

B. Contributions

Our work intends to provide a self-contained and easy-to-use software tool to facilitate the entire development cycle of MHCI systems. In particular, we extend the software framework presented in [12] and emphasize on solving the module reusability and software usability problems by introducing new mechanisms and graphical user interface (GUI) enabled software tool. More specifically, in contrast to the work presented in [12], the HCI^2 Workbench ('HCI^2' in its name is derived from the abbreviation of Human Centered Intelligent HCI [4]) embodies the following features:

1. Complete support of the development of highly flexible and reusable module packages. To increase module reusability, we distinguish between the concepts of module class and module instance. This refinement is implemented as a combination of SDK and other software tools. All are delivered in the HCI^2 Workbench. In this way, every step of the module development, including module program development, early stage debugging, and module packaging, is supported by the HCI^2 Workbench.
2. An easy-to-use centralized graphical environment facilitating module management, system configuration, module and system testing, module and system redistribution.

C. Organization of the Paper

The remaining part of this paper is organized as follows. Section II will introduce several design issues regarding the HCI^2 Workbench, including the underlying communication protocols, the concepts of module class and module instance, and the centralize system management scheme. The software implementation of our tool will be introduced in section III. In section IV, to help demonstrating its functionality, we will present a demo system developed using the HCI^2 Workbench. Finally, section V concludes this paper.

II. CONCEPTUAL DESIGN

This section describes several key design issues implemented in the HCI^2 Workbench. We will firstly introduce the runtime protocols adopted by the HCI^2 Workbench for data transport and runtime system management. Then we will highlight the concepts of module class and module instance, which play a vital role in achieving high module reusability. Last but not least, the centralized system management scheme, important for attaining software usability, will be explained.

A. Runtime Protocols

To facilitate reliable and efficient transport of high data rate message streams and achieve a high degree of system structure flexibility in the same time, we adopt the software framework proposed in [12].

The systems developed using the HCI^2 Workbench realize their structure using P/S architecture. In other words, instead of having the modules (which are primitive function units implemented as strand-alone programs) directly connected to each other, a number of channels are introduced to serve as 'connecters' between them [7] [9] [12] [14]. A channel is essentially a message relay, which simply routes every received (published) message to all of its subscribers. Furthermore, each channel is usually dedicated to deliver a single type of messages (e.g. video frames, audio samples, face box array, object position, and so on). A module may either subscribe or publish to a channel depending on whether it consumes or produces the messages in the channel's correspondent message type. The P/S architecture allows the modules to work independently from each other and brings natural support to complex (i.e., various looping structures) and dynamic (i.e., runtime-reconfigurable structures) system structures [14], hence improves flexibility on both module and system level.

In the HCI^2 Workbench, the P/S architecture is realized by a TCP-based runtime system management protocol as presented in [12]. In essence, the protocol defines several standard operations to support runtime structural changes including registration / unregistration of modules, establishment / cancellation of subscriptions, and creation / destruction of channels. In the HCI^2 Workbench, the protocol is extended to support more server-side operations, including compulsory module removal and server-side modifications to subscription relations, which are intensively used during the implementation of the centralized system management scheme.

The actual data messages are transported through shared memory for reliability and efficiency (in terms of message latency) [12]. A data transport protocol has been defined to regulate the process [12]. It adopts an asymmetric scheme, which distinguishes between message sources and sinks. In particular, the protocol defines two types of entities: message publishers and message consumers. Each message publisher corresponds to one channel and maintains a list of the channel's subscribers. When a message is published, the publisher will go through its subscriber list and write the message directly into every subscriber's inbox. On the other hand, each message consumer is allowed to subscribe to an unlimited number of channels. It allocates a local inbox in the shared memory and expects messages from all different sources. On the module level, each module maintains one message consumer and a number of message publishers, each corresponding to one channel that the module publishes to. In this way, the P/S communication is achieved uniformly across the system.

In addition to the simple message routing scheme, we extend the protocol to introduce built-in flow control. Since most algorithms used in a typical MHCI system are rather time-consuming, it is not unusual if a module lacks sufficient capacity to process all incoming data in time. In such case, the unprocessed messages will be queued in the module's internal buffer and cause it to expand indefinitely. This waste of memory is not only unnecessary but also harmful to the system's stability. To solve this problem, we insert a secondary buffer between each message consumer's local inbox and its output buffer, as illustrated in Fig. 1. The secondary buffer is organized into an automatically growing queue with a fixed maximum capacity. If the module's message retrieval rate cannot catch up with the input rate, the secondary buffer will be eventually piled full and consequently block the message consumer's internal message delivery thread. Hence, the message source will be forced to wait before subsequent messages can be successfully published. In this way, the message rate in each processing pipeline will converge to an optimal level over time. Hence automatic flow control is achieved.

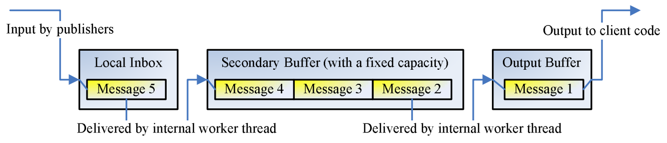

Input by publishers
Local Inbox
Message 5
Secondary Buffer (with a fixed capacity)
Message 4
Message 3
Message 2
Output Buffer
Message 1
Output to client code
Delivered by internal worker thread
Delivered by internal worker thread

Figure 1. The internal buffer structure of a message consumer.

B. *Module Class and Module Instance*

Although the P/S architecture eliminates inter-module dependencies and improves flexibility, it does not necessarily lead to reusable modules. One key issue that still resides is the structural dependency problem. Consider the following case. Suppose a face detection module and an object tracking module both take input from one video stream, which may be named 'Video_In' in both modules' subscription list. In this case, if the face detector uses grayscale image while the object tracker requires color information, their actual requirement on the 'Video_In' channel's message format will be different. Consequently, since the uniqueness of channel ID is necessary, these two modules will not be able to co-exist in the same system without modification to their source code. The hard-coded channel ID in a module's implementation generates unnecessary dependency to its local environment, hence limits its reusability.

A module developed using [9], [10], or [12] may contain even more hard-coded information, including module ID, algorithm parameters (e.g. classifier coefficients), input / output message specification (e.g. video resolution) and so on. Each additional hard-coded entry poses more limits to the scenarios in which the module may be used. In fact, this lack of generality can be traced back to CDM. CDM proposes a development procedure for systems built from scratch. It does not give much concern to module reusing, hence may result in highly task-specific modules if being followed blindly.

As a remedy to this problem, we refine the original concept of module into two related terms, which are the module class and the module instance. The basic idea is that a developer should design his / her module in its most general form and distinguish between task-specific requirements and generic specifications. This concept is derived from the methodology of object-oriented design, hence the similar terminology.

1) Module Class

A module class consists of the generic implementation of a function unit. It usually contains a group of files including the algorithm implementation (the portal program), a human-readable help document, instance-invariant data / binary files, and so on. Hence, sometimes a module class is also called a module package. In addition, the module class further provides a template specifying which task-dependent information it needs to create an instance for specific use.

To store both the content file list and the instantiation template into what we call a module description file (MDF), a standard XML semantics is defined. In particular, each module description file encapsulates the following aspects of information:

1. Content file list, which specifies the files included in the module package. We define three types of files: the portal program (also known as the module program), the help document and the miscellaneous dependencies (e.g. DLLs, data files, etc.). Among these files, the module program is the executable file which will be called when any of the module class's instances is activated, hence is necessary in all module classes. All other files are optional. The content file list plays an important role in module class redistribution.

2. Input / output (I/O) specification, which declares the configuration of the input / output channels. Specifically, this section defines which channels the module may subscribe to and which channels it may publish to. Nonetheless, instead of using a fixed ID, each channel is identified by its locally-unique (within the scope of the module class only) alias. This design is based on the experience that although the actual channel IDs may differ, a module's I/O configuration is often instance-invariant. During instantiation, the channel aliases will be mapped to the IDs of the channels that are actually used in the particular system. By using this channel mapping mechanism, the modules no longer display structural dependency on their local environment, hence may be used freely in any system.

3. Parameter template, which gives a specification to the rest of the task-dependent information. In particular, each entry in the parameter template describes one parameter by its name, type, default value, and range. Three basic parameter types are defined, which are integer, real number, and text. To cope with composite data-types, we define an additional parameter type, namely, the raw byte string, which can be used to represent any serializable data-type. Since XML only supports printable characters, the raw byte string parameters are stored as Base64 encoded strings in practice.

Note that we often use the term module as an abbreviation of module class in this paper, unless the discussion is within the scope of some particular system, in which case it stands for module instance.

2) Module Instance

A module instance consists of a function unit within the scope of a particular system. Thus, in addition to the generic data and operations provided by its module class, a module instance also needs to have its own task-specific information. This information is stored in its correspondent module configuration file (MCF), which also follows a standard semantics defined over XML. Specifically, a module configuration file often contains the following fields:

1. Registration information, which specifies the module instance's ID, inbox size, and its secondary buffer capacity.
2. Channel mappings, which configure the mapping relations between the channel aliases defined in the module class description and the actual channels used to deliver messages in their expected types in the current system.
3. Parameter values, which are the values given to the parameters specified in the module class's description file.

With the task-specific information given by the MCF, a suitable instance may be easily derived from a given module class to work in any particular environment (system) without requiring explicit adaptation, hence the convenience in module reusing.

3) Centralized System Management

While the distributed nature of the P/S architecture brings flexibility, as a side effect, it may also lead to potentially counter-intuitive and cumbersome system integration and testing process. The problem is that it would be difficult for the developers to perceive a system as an integrated entity with a comprehensible internal structure if all modules within the system needed to be executed as standalone applications (as was the case in [9], [10] and [12]). In that case, the system would be hard to build, configure, test and redistribute.

Hence, to improve user experience, it is essential to have a working environment that creates an 'expression' of centralized control. More specifically, in order to be able to conveniently construct a system from existing module classes, an explicit system representation method is vital. This is achieved cooperatively by the module warehouse and the system configuration files.

As its name suggests, the module warehouse serves as a global storage of the module classes used as the basic system building blocks. Inside the module warehouse, the module packages are arranged into a folder structure mimicking a linear table, which allocates a separate folder for each package. The module warehouse also maintains a content list in the form of a XML file. This simple structure allows a module class to be easily imported and / or exported, which hardly requires more operations than copying a folder.

The system configuration files (SCFs) are used to represent the actual systems. Since the module classes are stored in the module warehouse, a system configuration file only needs to save the system structure. In particular, it specifies the channels, the module instances and the subscription / publication relations between them. Similar to the module description files and the module configuration files, we define a standard XML semantics for the system configuration files as well. In essence, a system configuration file roughly consists of the concatenation of the module configuration files specifying the system's constituent module instances plus a list of the channels it uses. Because each system configuration file serves as a self-contained repository for all of the information defining a given system, the procedure for system reconfiguration and redistribution are reduced to mere file operations.

By using the module warehouse and the system configuration files, building a working environment facilitating centralized system management becomes a trivial task. An intuitive visual display (in the form of a dynamic block diagram) of a given system' structure can be easily produced by parsing its correspondent system configuration file. Moreover, the system structure display can be extended into a control interface as well, which may support system reconfiguration through simple GUI commands that are mapped to proper file editing operations. Activating a system or some of its module instances is also rather straightforward. The working environment merely needs to split the system configuration file into module configuration files and pass them as command-line parameters while executing the appropriate portal programs delivered by the module packages stored in the module warehouse.

III. Software Implementation

The HCI^2 Workbench is implemented as a self-contained software development tool. It is further divided into three parts: the SDK for module development, a number of GUI-enabled software tools aimed at module development and packaging, and the system construction workbench providing an integrated graphical working environment for module class management, system construction, and testing.

Due to the shear size of our code base, this section will not be able to cover much implementation detail. Instead of that, we will focus on the high-level description of the functionality of the tool we provided and how it may be used to facilitate a typical system development process.

A. Module Development SDK

The module development SDK is one of the key components of the HCI^2 Workbench. It encapsulates the implementation of all operations defined in the runtime protocols and the file handlers for dealing with the various configuration / descriptions files and exports a compact set of programming interfaces through a small number of C++ virtual classes (also known as interface classes).

The source code of all classes provided in the SDK are freely available and well-documented, hence can be easily customized. Alternatively, for most applications which require no customization, we also provide a binary package of this

SDK. This package consists of 4 precompiled statically-linked libraries, each of which houses a branch of closely related classes, as described below.

1) *Exception Hierarchy*

To represent various runtime errors ranging from mild conditions such as communication timeout or invalid parameter value to severe problems including system call failure and connection loss, we derive a number of hierarchically organized custom exception classes from the STL's std::exception class. Although this exception hierarchy was primarily designed for our module development SDK, it also has a good coverage to most of the error conditions which may arise in many other types of applications. Thus we package these classes into a separate library, which may be reused in other projects.

In addition to the 'usual' textural description, our exception hierarchy further delivers the following features:

1. Catch stack: Each exception maintains a catch stack to store its trace through the error handling blocks. In particular, the stack grows when the exception object is caught and re-thrown by a catch block. If used properly, the catch stack should more or less resemble the function call stack (but in reverse order), which may help developers to derive more pertinent error handling methods.
2. Inter-thread exception handling: Through appropriate cooperation from client code, the exception hierarchy allows an exception thrown from one thread to be handled within another thread. The key idea is to maintain a data structure in local heap as a central repository for all exceptions regardless their origin thread. An additional checking step is then built into the catch blocks to enable asynchronous handling of such exceptions.

2) *Basic Communication Adapters*

The basic communication adapters implement the runtime protocols described in subsection II.A. This group of classes is further divided into two sets which encapsulate the server-side operations and the module-side operations, respectively. Each of the two sets of classes exports a single virtual class as the programming interface.

These classes reuse the reference middleware implementation presented in [12], but with the following improvements:

1. They adopt the general design pattern of interface / implementation separation (which is also used by most of classes except of those in the exception hierarchy) [15]. In particular, each class is implemented as a pair of C++ classes, which consist of a virtual class specifying the public function interface and a concrete class derived from it to provide the actual implementation. Hiding implementation details (in particular, private members) reduces inter-class dependency and accelerates the compilation process [15]. More importantly, it eliminates all references to operating-system-dependent data-types / operations from the SDK's public interface. Hence, if the HCI^2 Workbench is to be migrated to other operating systems (other than Windows) in the future, the source-code level cross-platform support to the client programs will be inherently granted.
2. Finer encapsulation granularity is adopted. In particular, we implement the basic TCP messaging routines, the shared memory data transport protocol, and the runtime system management protocols as separate classes (or groups of classes). In this way, these building blocks may be upgraded independently when needed. This modification also allows us to refine locking granularity by substituting the exclusive locks with reader-writer locks and hence reduces waiting time in the multi-threaded operations.
3. There is no limit in the number of subscribers. By implementing our own multilayer semaphore waiting operation, a structural limit (which was 64) in how many subscribers a given channel may have is no longer needed.

3) *File Handler Classes*

As their name suggests, these classes are implemented to parse and generate the various configuration / description files on which the HCI^2 Workbench relies. In addition to the basic file operations, these classes also provide built-in support to validity check and error recovery on semantic level. For instance, when dealing with an erroneous module configuration file, the parser class will firstly attempt to recover as many errors as possible (e.g. inconsistent channel mappings, absent fields, etc.) before reporting parsing failure to the client code. This design improves error tolerance and consequently the overall robustness on both the module and the system level.

4) *Integrated Plug-in Component*

This component integrates the module configuration file parser and the basic module-side communication adapter to provide a compact programming interface for the development of most module programs. The integrated plug-in component exports its operations through a single virtual class, which should be the only class a typical module developer needs.

Functionally, the integrated plug-in components exports a similar set of operations to that of the basic module-side communication adapter. In particular, it delivers a number of functions for publishing / retrieving messages, establishing / cancelling of subscriptions, and so on. Nevertheless, the actual implementation of these functions is different. The operations exported by the plug-in component embody built-in support to channel mapping. In other words, all functions involving channels take channels aliases instead of channel IDs as their parameters. Internally, the passed-in aliases are firstly translated into the actual channel IDs (with respect to the channel mappings specified in the current module instance's module configuration file) before the protocol operation is invoked. The translation is also performed for returned information, but in reverse direction. Specifically, the channel ID field in the retrieved data messages and system notifications are replaced with their correspondent channel aliases. Furthermore, the module insulation is strengthened as well. For instance, the integrated plug-in component no longer allows client programs to query the subscriber list of a given channel or to create / destroy channels remotely.

With these features, the integrated plug-in component eliminates all explicit reference to specific channels and / or module instances. Hence it prevents the introduction (either accidental or deliberate) of structural dependency of any kind into the modules, and consequently guarantees a high degree of module reusability.

B. *Module Packaging Tools*

While the module development SDK facilitates the development of a module's portal program, we provide additional software utilities to cover other steps involved in a typical module development process. These tools are collectively known as the module packaging tools, which are explained below.

1) *Module Description File Editor*

The module description file editor is essentially a GUI-enabled file editor program. It allows developers to edit the specification of a module class. Internally, the file parsing and generation services are provided by the file handler classes included in the module development SDK. The MDF editor's graphical display has a direct correspondence to the logical representation of the module description, thus allows developers to skip the syntactical and semantic notations altogether. Moreover, the MDF editor enables developers to create sample module configuration files complying with current module class description in one mouse click, which may greatly simplify the debugging process.

2) *Module Configuration File Editor*

The module configuration file editor is another GUI-enabled file editing tool. It is similar to the module description file editor in all aspects except of the targeted file type. Although users may never need to manually create a module configuration file for system execution as the module activation process is fully automated, sample module configuration files may still be useful during early stage module program debugging. This is especially true if a third-party debugger is to be used. In that case, a developer may use the module configuration file editor to generate a sample file and pass it as a command-line parameter while executing the module program (which always requires a MCF to run) from the debugger's environment.

3) *Basic Server*

The basic server is a GUI shell built around the server-side communication adapter implemented in our SDK. Albeit simple in appearance, this tool is a fully functional server program which can be used to support any system, but in a 'traditional' way, that is, requiring the user to start and configure the server program first and then manually executes all module programs with appropriate settings. In particular, the basic server displays the runtime system structure through a channel list and a module (instance) list and provides command for all server-side operations defined in the runtime system management protocol. This utility is also an auxiliary tool for the debugging process. It may also be used as a substitution to the system construction workbench in rare cases where the centralized control scheme is somehow not favored.

4) *Module Program Debugger*

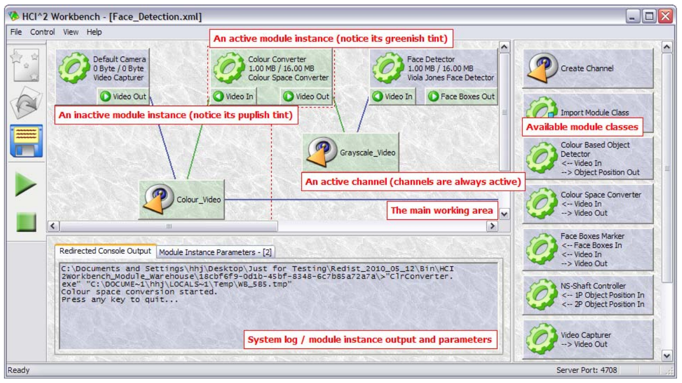
The module program debugger is a console application which is specifically provided for debugging GUI-enabled module programs. Because of the console output redirection mechanism implemented in the system construction workbench, every module program, regardless of its category (i.e., whether it is a console application or a GUI-enabled Win32 application), is allowed to output runtime status using the standard STL streams. Nonetheless, the textual output produced by a GUI-enabled module program is often invisible due to lack of console window if the program is executed alone. Hence, to allow this kind of programs to be debugged in a 'realistic' environment that is fully compatible with the system construction workbench, the module program debugger is built. Essentially, this tool implements the same output redirection mechanism as that of the system construction workbench and displays the redirected content in a console window, thus allows the module program's console output to be examined during early stage debugging within the program's indigenous environment.

C. *System Construction Workbench*

The system construction workbench provides an integrated working environment for module class management, system configuration, and system / module testing and redistribution. Its GUI is shown in Fig. 2. As can be seen, the main window of the tool is split into three areas, which are the system editing area (top-left), the status area (bottom-left), and the module warehouse display area (right).

The system editing area essentially displays the current system structure using a dynamic block diagram and allows developers to easily modify the structure using mouse clicks only. In Fig. 2, the system loaded in the workbench is a face detection demonstrator that uses Viola-Jones face detection algorithm [16]. As the figure illustrates, all channel and module instance are displayed as a rectangles showing their key configuration including ID, module class name, inbox size, secondary buffer capacity, and I/O configuration (the last 4 entries are for module instances only). Different icons are used to distinguish the two types of entities. The channel mappings are represented by the lines connecting the channels to their mapped channel aliases shown inside the module instance representations. Furthermore, the runtime information is also displayed in the diagram using a consistent color coding scheme. For instance, as shown in the Fig. 2, active module instances are rendered in green tint while the inactive ones are in purple.

The system editing area also provides many commands for system reconfiguration and testing. For instance, the channel mappings may be established or cancelled by selecting the appropriate popup-menu items. Channels and module instance may be renamed (or reconfigured), added or removed easily and they can be dragged around by mouse to produce better graphical layout. To test the system, a developer may use the 'Activate All' menu command to start the whole system in a single click. Alternatively, the workbench also allows the developer to activate or deactivate each module instance separately through its own popup-menu. To maximize flexibility, it allows the developer to reconfigure a part of the system or update some module classes while keeping the remaining parts of the system active.


HCI^2 Workbench - [Face_Detection.xml]
File Control View Help
An active module instance (notice its greenish tint)
Default Camera 0 Byte / 0 Byte Video Capturer
Video Out
Colour Converter 1.00 MB / 16.00 MB Colour Space Converter
Video In
Video Out
Face Detector 1.00 MB / 16.00 MB Viola Jones Face Detector
Video In
Face Boxes Out
An inactive module instance (notice its puplish tint)
Grayscale_Video
Colour_Video
An active channel (channels are always active)
The main working area
Create Channel
Import Module Class
Available module classes
Colour Based Object Detector <-- Video In --> Object Position Out
Colour Space Converter <-- Video In --> Video Out
Face Boxes Marker <-- Face Boxes In <-- Video In --> Video Out
NS-Shaft Controller <-- 1P Object Position In <-- 2P Object Position In
Video Capturer --> Video Out
Redirected Console Output
Module Instance Parameters - [2]
System log / module instance output and parameters
Ready
Server Port: 4708

```
C:\Documents and Settings\hhj\Desktop\Just for Testing\Redist_2010_05_12\Bin\HCI
2Workbench_Module_Warehouse\18cbf6f9-0d1b-45bf-8348-6c7b85a72a7a\>"ClrConverter.
exe" "C:\DOCUME~1\hhj\LOCALS~1\Temp\WB_5B5.tmp"
Colour space conversion started.
Press any key to quit...
```

Figure 2. Main window of the system construction workbench.

The status area is used to display the detailed information regarding the currently selected object. The entries vary depending on the selected object's type. If a channel is selected or if no selection is presented, a log of system events will be displayed instead. In particular, it will show a history of channel creations / destructions, module instance registrations / unregistrations, and establishments / cancellations of subscriptions. Otherwise, if a module instance is selected, the status area will expand to include two tab pages, one for the module instance's redirected console output and the other one for its parameters, which can also be adjusted easily with appropriate parameter editing dialogue boxes.

The module warehouse display area provides a user interface for module class management. In essence, it lists all available module classes, which can be instantiated, imported, or removed through appropriate popup-menu commands. In addition, the workbench also allows a developer to export or update existing module classes. Among these two functions, module updating is particularly useful for module testing. One key feature is that the workbench allows a module class to be updated while it is still in use by the current system. In that case, the corresponding module instances will be validated against the updated module description and automatically revised if necessary. This feature enables the developer to test and fine-tune a module program efficiently, without being required to restart or even reconstruct the entire system after each tiny modification.

Last but not least, the system construction workbench also facilitates redistribution of module classes and systems. On module class level, the exporting function is obviously useful for redistribution. In addition, because the module warehouse is arranged in a self-contained folder structure, one may also copy the top-level folder to another machine to ship all module classes in a single round. Since each system is fully described by a system configuration file, redistributing a system is trivial, merely requiring the developer to copy the file in question to the target machine.

IV. GAMEGAME: A DEMO SYSTEM

To demonstrate the proposed tool and to provide a tutorial example for potential users, we have developed an open-source demo system called the CamGame, which is shown in Fig. 3. CamGame system enables one or two players to play a computer game using low-cost camera(s) (e.g., USB Webcams) and hand-held marker(s) (any brightly-colored rigid object may be used) instead of keyboard and mouse.

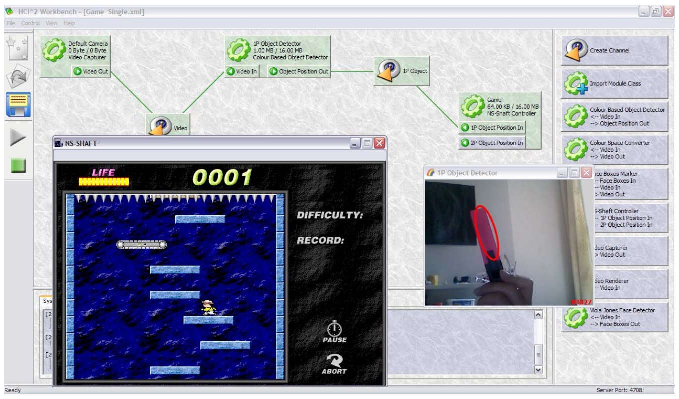


Figure 3. The CamGame system in single-player mode.

The system requires three processing stages: video capturing, marker tracking, and command mapping. We develop a module class to accomplish each of the three tasks.

Video capturing is performed by the video capturer, a console program which captures real-time video stream using DirectShow and publishes the frames to an output video channel.

Many algorithms exist for object tracking under various conditions. Since this system is a demonstration to the HCI^2 Workbench instead a particular development in the field of computer vision, we intend to keep the actual processing as simple as applicable. Thus we use the mean-shift algorithm described in [17], which is relatively easy to implement while yielding good tracking results. In practice, the marker tracker is built as a module that takes input from a video channel and displays the images in its main window with the tracking results illustrated by an overlapped ellipse. It requires the player to initialize the tracking process by selecting the marker in the first video frame of the input. Once initialized, the module will continuously produce tracking results (including the marker's position, size, and orientation) and output that to the appropriate channel.

A command mapper is used to map the input marker positions into messages to control the actual game (note that we did not develop a game but a game controller). Since the assumption is that the game itself is only controlled by two keys, which are 'left' and 'right', the mapping scheme is rather simple, which merely maps the marker position to 'left' and 'right' according to its horizontal location.

The actual structure of the system is illustrated in Fig. 3. It consists of a of 3 module instances, one of each class introduced above, connected by 2 channels. This version is used to play the game in a single-player mode. We also create another version allowing the game to be played in cooperative, two-player mode. The structure of this modified version is shown in Fig. 4. In particular, an additional instance of the marker tracker and a new marker-position channel are added to the system to produce control messages for the second player.

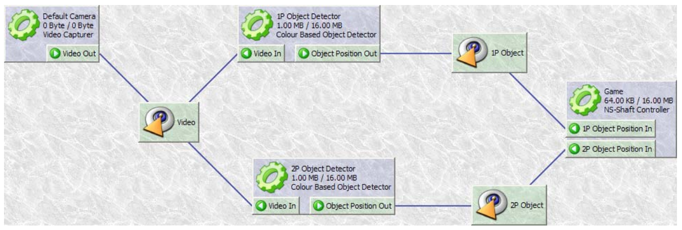


Figure 4. The CamGame system in cooperative-mode.

V. Conclusion

Based on the software framework presented in [12], we have proposed a self-contained and easy-to-use software tool to facilitate the development of multimodal human-computer interaction systems. This software tool, which is called the HCI^2 Workbench, consists of a combination of SDK and GUI-enabled utilities to provide complete support for the system development procedure, spanning from module program development, system construction to testing and redistribution.

Internally, the HCI^2 Workbench adopts a P/S architecture to support flexible system structures while uses a shared-memory based data transport to guarantee reliable and low latency delivery of high data-rate message streams. Moreover, to improve module reusability, we have refined the concept of module into two related terms, which are the module class and the module instance. Based on the concept of module warehouse and the XML semantics for system configuration files, we have proposed a centralized system management scheme, which greatly simplifies system construction, testing, and redistribution.

In practice, the tools included in the HCI^2 Workbench are divided into three parts, which are the SDK for module program development, the auxiliary tools for module debugging and packaging, and the integrated working environment facilitating module class management, system integration, system / module testing, and redistribution through an intuitive GUI.

To better demonstrate the HCI^2 Workbench and to provide a tutorial example, we have developed a simple interactive system, called the CamGame, which enables players to play a computer game using hand-held markers and ordinary low-cost webcam instead of keyboard and mouse.

The redistribution package of the HCI^2 Workbench is now available at: http://ibug.doc.ic.ac.uk/resources.

References

[1] M. Pantic, and L. J. M. Rothkrantz, "Towards an affect-sensitive multimodal human-computer interaction", Proceedings of the IEEE, vol. 91, no. 9, pp. 1370-1390, September 2003.

[2] A. Jaimes, and N. Sebe, "Multimodal human–computer interaction: a survey", Computer Vision and Image Understanding, vol. 108, no.1-2, pp. 116-134, 2007.

[3] M. Pantic, A. Pentland, A. Nijholt and T.S. Huang, "Human computing and machine understanding of human behavior: a survey", Artificial Intelligence For Human Computing, T.S. Huang, A. Nijholt, M. Pantic and A. Pentland, Eds. Springer, Lecture Notes in Artificial Intelligence, vol. 4451, pp. 47-71, 2007.

[4] M. Pantic, A. Nijholt, A. Pentland, and T. Huang, "Human-centred intelligent human-computer interaction (HCI^2): how far are we from attaining it?", Int'l Journal of Autonomous and Adaptive Communications Systems, vol. 1, no. 2, pp. 168-187, 2008.

[5] Z. Zeng, M. Pantic, G.I. Roisman and T.S. Huang, "A survey of affect recognition methods: audio, visual, and spontaneous expressions", IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 31, no. 1, pp. 39-58, 2009.

[6] A. Vinciarelli, M. Pantic, H. Bourlard, "Social signal processing: Survey of an emerging domain", Image and Vision Computing, vol. 207, no. 12, pp. 1743-1759, 2009

[7] L. Maat, and M. Pantic, "Gaze-X: adaptive affective multimodal interface for single-user office scenarios", Artificial Intelligence for Human Computing, T. S. Huang, A. Nijholt, M. Pantic, and A. Pentland, Eds. Springer, Lecture Notes in Artificial Intelligence, vol. 4451, pp. 251-271, 2007.

[8] "MSDN: DirectShow (Windows)", Dec. 4, 2008. [Online]. Available: http://msdn.microsoft.com/en-us/library/dd375454(VS.85).aspx.

[9] "Communicative Machines: Pscylone", 2007. [Online]. Available: http://www.cmlabs.com/psyclone/.

[10] "Apache ActiveMQ", 2009. [Online], Available: http://activemq.apache.org/.

[11] J-Y. Lawson, J. Vanderdonckt, and B. Macq, "Rapid prototyping of multimodal interactive applications based on off-the-shelf heterogeneous components", Adjunct Proceedings of the 21st Annual ACM Symposium on User Interface Software and Technology, pp. 41-42, 2008.

[12] J. Shen, and M. Pantic, "A Software Framework for Multimodal Human-Computer Interaction Systems", Proceeding of IEEE Inrernational Conference on Systems, Man and Cybernectis, pp. 2038-2045, 2009.

[13] M. Pantic, R.J. Grootjans, and R. Zwitserloot, "Fleeble agent framework for teaching an introductory course in AI", IADIS International Conference Cognition and Exploratory Learning in Digital Age, pp. 525-530, 2004.

[14] K. R. Thorisson, H. Benko, D. Abramov, A. Arnold, S. Maskey, and A. Vaseekaran, "Constructionist design methodology for interactive intelligences", AI Magazine, vol. 25, no. 4, pp.77-90, 2004.

[15] S. Meyers, "Minimize compilation dependencies between files". Efective C++: 50 Specific Ways to Improve Your Programs and Designs 2nd Edition, pp. 140-148, Addison Wesley, October, 1997.

[16] P. Viola, and M. J. Jones, "Robust real-time face detection", International Journal of Computer Vision, vol. 57, no. 2, pp. 137-154, 2004.

[17] D. Comaniciu, V. Ramesh, P. Meer, "Real-time tracking of non-rigid objects using mean-shift", IEEE Conference on Computer Vision and Pattern Recognition, vol. 2, pp. 438-445, 2000.