CHI: A General Agent Communication Framework
Laurence R. Phillips, Steven Y. Goldsmith, & Shannon V. Spires

Sandia National Laboratories
Albuquerque, NM 87185
lrphill, sygolds, svspire@sandia.gov
505-845-8846, -8926, -4287

Abstract: We have completed and exercised a communication framework called CHI (CLOS to HTML Interface) by which agents can communicate with humans. CHI follows HTTP (HyperText Transfer Protocol) and produces HTML (HyperText Markup Language) for use by WWW (World-Wide Web) browsers. CHI enables the rapid and dynamic construction of interface mechanisms. The essence of CHI is automatic registration of dynamically generated interface elements to named objects in the agent’s internal environment. The agent can access information in these objects at will. State is preserved, so an agent can pursue branching interaction sequences, activate failure recovery behaviors, and otherwise act opportunistically to maintain a conversation. The CHI mechanism remains transparent in multi-agent, multi-user environments because of automatically generated unique identifiers built into the CHI mechanism. In this paper we discuss design, language, implementation, and extension issues, and, by way of illustration, examine the use of the general CHI/HCHI mechanism in a specific international electronic commerce system. We conclude that the CHI mechanism is an effective, efficient, and extensible means of the agent/human communication.

Introduction
Communication underlies and forms a context for community. Although the World-Wide Web (WWW or just “the Web”) forms an extensive digital community, communication mechanisms are often laboriously hand-built, application-specific and not very flexible. Agents promise to alleviate this difficulty by providing personal guides that either understand humans, understand their environments, or both; at a minimum, such agents will understand how to communicate using the available media: e-mail, fax, and most importantly the Web.

The key to making this work in practice is the establishment of appropriate abstraction barriers and the factoring of capability into more-readily maintained components. In particular, we advise against agents with monolithically embedded communication mechanisms. A plug-in or mix-in capacity to use whatever media may be available is more robust and easier to manage and maintain. This approach encapsulates the means of communication and keeps it independent of the agent’s reasoning means. The means—methods specialized on multiple inputs, classes modifiable at runtime, multiple inheritance—are available in modern dynamic languages. This approach also scales well across the important dimension of language type. We examine this issue during discussion of the international electronic commerce system, in particular how information from several sources in several private business dialects is incorporated into the mechanism.

We have developed CHI (CLOS-to-HTML Interface), a class/method hierarchy that enables communication between agents and humans. The information moving between the user’s web browser and the agent during communication consists of objects transformed into Hypertext Markup Language (HTML). These objects are “carriers” of information and allow the user to have the illusion of direct interaction with the agent. The agent, on the other hand, relies on a set of

---

1 This work was performed at Sandia National Laboratories, which is supported by the U.S. Department of Energy under contract DE-AC04-94AL85000
DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, make any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.
DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.
independent performatives, such as those expressible in KQML ([Finin et al.], [Finin and Labrou], and [Finin et al.]). The conversion between an internal object representation and KQML would allow agents to communicate with one another in much the same way that CHI enables communication between agents and humans. This paper focuses on agent/human communication; however, we discuss extension to other communication channels in the “Extension to other languages” section.

How CHI works

From the user’s point of view, the user hits the “submit” button of a form or clicks an anchor, the browser says “Host contacted; waiting for reply ...”, and shortly a (new) page is displayed on the user’s browser.

The agent with whom the user is communicating must receive the message sent by the user (which implies that some connectivity must exist between users and agents in multi-user multi-agent situations), generate a response specifically for that user (which implies that the message must be identified as having come from the user), and send the response back to the user (which implies that the agent must have some connection with the user).

CHI must therefore enable the agent to transmit a page to the user’s browser in response to each form submission or anchor click (POSTs and GETs, in HTTP terms). In this paper, we use the term “post/get” to refer to any such transmission.

This is accomplished by the following cycle, illustrated in Figure 1:

1. CHI creates and initializes a page object. The appropriate class is instantiated based on the class of the message object being transmitted. Message components are similarly converted into elements in the page object.

2. CHI translates the page object (and its components) into an HTML stream.

3. CHI entrains the ACGI mechanism to the intended recipient’s input channel.

4. The recipient takes some action that returns a message to the CHI application (The server receives the GET/POST and sends it to the CHI application via the ACGI mechanism).

5. CHI extracts the identifier from the message and uses it to acquire the page object from which the HTML page was generated.

6. CHI places information from the message into the page object.

7. CHI executes relevant methods and functions, dispatching based on the class of the page object, to produce the next page object. Interaction with a session object can occur.

8. CHI links the old page object and the new page object if necessary.

9. The cycle begins again with the new page instance.

Developing applications using the CHI methodology

CHI is a set of classes and methods. Instances of CHI classes are able to render themselves as HTML. The highest-level objects result in pages or frames whose components contain the information being communicated. An agent can access this information at will. Nested objects are called recursively; each generates its own HTML. Nesting is maintained: Tables contain rows,
class of interface, not class of human

CHI method dispatches on two classes

Create page (_ , _)

Create page

CHI instantiates page object

Note the information containers are exclusive-dependent to the message object

CHI extracts tagged data and modifies the object at the OID. Note that this is the same object that's in the agent's message object.

△ = object class

1. Agent sends performative “send this message (object) to this human (object)"
2. CHI creates page object with embedded message objects
3. CHI generates HTML with OID (object identifier) and tags
4. Human modifies information and returns page to CHI
5. CHI reads information from text and registers with object using OID and tags
6. Modifications to objects are instantly visible to agent

Figure 1. The CHI information cycle from Agent to human and back
which contain cells, which contain paragraphs, etc. The CHI-generated HTML contains tags and names that enable CHI to identify and update the source objects when information from a CHI-generated page is returned to the server. Input sites (textboxes, radio buttons, etc.) in the HTML must be named uniquely within each page (a utility function reports duplicates) so that CHI can retrieve the individual information values and update the original objects.

The objectified version of each HTML input site has a slot called `my-field`. The field slot automatically receives an instance of class `field`. Field objects support an update/changed?/retrieve protocol and are the application side's view of the interface. The agent uses these objects to register its internal data in the page. In essence, a page instance is handed to the agent at “maketime” and the agent runs a simple utility over the object which inserts its own field objects into the `my-field` slots of the page. Correspondence between the field objects and the input sites is based on their well-known names. Name correspondence must be maintained and is the sole connection between the user interface and the application. The name and nature of the named elements is the entirety of what must be decided upon between the interface designer and the application designer.

CHI automatically maintains program state and provides continuity for multiple simultaneous user sessions (see Figure 2) by linking together dynamic objects that represent pages and sessions. This solves the problem of operating a stateful application with a stateless browser (discussed in the “Preservation of state in a stateless environment” section below). CHI supports workgroup computing on the Web because the CHI process maintains multiple user states simultaneously and non-synchronously.

Session objects are not provided by CHI but are enabled by the state-preserving nature of CHI. If sessions are needed, the application designer must provide session objects that accommodate the following protocol:

1. A session can be created;
2. A session has a starting time and an ending time;
3. A session may contain objects;
4. Sessions can be retrieved once they have ended;
5. Objects contained in a session can be retrieved;
6. A session and its contents persist after the session ends; and
7. A session is accessible only via secure interface.

The session object is the primary vehicle for building, storing, and retrieving persistent information within an application framework. In particular, page objects can be stored in the session. The session enables objects outside the client/server relationship to be owned by specific (possibly human) agents. The resultant identified ownership enables control of access to both the objects and their lifecycle protocol (create, modify, destroy).

The task of designing an agent communication interface consists in creating a set of classes, initialization mechanisms, and methods that “glue” the agent and the user’s Web browser together. This ultimately engenders two primary tasks: (a) Define the class structure of the objects that constitute the interface and (b) Write the runtime methods that:

1. Translate information from the user into a syntax that the agent understands,
2. Send a performative (or performatives) to the agent that contain the translated information,
3. Receive the agent’s response,
4. Convert the agent’s response into HTML, and
5. Return the HTML to the user.

The CHI environment realizes this protocol. The CHI environment provides base classes and functionality. The designer creates specialized subclasses of the CHI classes containing information specific to the agent’s needs. The designer also adds look and feel, task-organization, and so on.
CHI is essentially a class library. Instances of CHI classes can convert themselves into HTML using the widely-inherited and specialized VIEW-AS-INTERFACE method. Among the interface designer's tasks is defining CHI-based page objects that objectify the HTML interface. This can be done from first principles using standard constructs; however, this is a tedious, complex, and unforgiving process. To assist, we have developed a software mechanism called HCHI (HTML to CHI) that does exactly what its name implies: HCHI automatically converts any HTML page or frame into two pieces of executable code: (a) a class definition that inherits from existing CHI base classes and (b) a specialized version of the generic MAKE-OBJECT initialization function that creates the appropriate nested CHI instances within the page object when it is created. The interface designer exercises HCHI on every HTML page in the application (actually, on one and only one example of every type of page) to produce the corresponding class definition and MAKE-OBJECT method. In this paper we reserve further discussion of HCHI and focus on the CHI mechanism.

Having by some method produced page classes and initialization functions that produce CHI-based page instances, the interface designer then writes an ENSURE-NEXT-PAGE method for each interface page class. The job of the ENSURE-NEXT-PAGE methods is to return a properly fleshed-out instance of the appropriate page class in response to every input that could come into the server application (and thence, ultimately, to the agent or agents) from its corresponding screen. HCHI constructs these classes automatically from example HTML page files and. These response page instances are automatically converted into HTML by their respective VIEW-AS-INTERFACE methods for consumption by the client browser. Figure 2 shows the mechanical view of CHI in operation (as opposed to the conceptual view shown by Figure 1).
The designer of a CHI application is almost entirely absolved of any contact with HTML code. Once the interface has been defined (i.e., once an example of every possible page has been built; we recommend using any of several off-the-shelf HTML editing tools) and all input sites named, the interface designer runs HCHI to generate the required CHI-based classes and provide direct named-variable access to the data provided by the input sites. These define the structure of the browser pages that the user will see. The behavior that produces subsequent pages is based on data in the POST messages from the HTML page and on the ENSURE-NEXT-PAGE method for the page. The actual page contents are determined and page instances are constructed at run time, thus turning the Web into a dynamic computing environment.

Interfaces with many interactive and inter-related Web pages can be created in a few hours. The HCHI mechanism registers each relevant object in an agent’s internal object-oriented framework with HTML elements generated at runtime by CHI. Registered objects are then automatically updated based on GET/POST messages from the client. A new web page is created each time the server needs to send and/or receive information to a user. Certain dynamic characteristics are provided free-of-charge by CHI; for example, the number of rows in a table is determined at runtime and is therefore entirely flexible. Similar dynamics determine the contents of radio button sets, groups of checkboxes, “submit” button labels, selection menus, object color, text values, etc.

This process is automatic; the agent merely receives message notification and updated objects without further reference to HTML or HTTP syntax, which is fully encapsulated within CHI. Formulation of a response is also entirely internal to the agent and independent of HTML. At output time the agent, from its point of view, merely emits its chosen response to the relevant user; CHI generates a new web page instance dynamically, so there is never an html file anywhere in the system (with the possible exception of an initial template, which can also be generated by an agent at the initiation of a contact).

Extension to other languages

CHI provides a set of foundation classes that represent pages, forms, and widgets. Instances of these classes translate themselves into HTML at run-time for Web consumption. CHI classes are hierarchically organized and available to the designer for specialization, along with their inherited methods. We do not expect to rewrite CHI in some other language; however, rapid changes are expected in the communication language realm, witness HTML 3, XML, and so on. We do expect to be able to increase the envelope of agent response with respect to the Web by dynamically generating Java and/or JavaScript.

Updating CHI itself to support changes in HTML or HTTP, or extending CHI to wholly different interface languages, is conceptually straightforward because the language specifics are factored into the VIEW-AS-INTERFACE methods specialized on the interface class. This also means that different versions of HTML and different interface languages can be supported simultaneously by a single running CHI server.

The link between the agent’s own internal language and that of the recipient resides in these multi-methods, that is, methods specialized on multiple classes. The specializing classes are the output message class and the recipient’s interface type (see Figure 1). Invariant semantic content is embedded into a stream of the appropriate syntax as the message object is converted by the appropriately specialized method regardless of the language.

At this time, agents equipped with CHI can communicate only with humans. The multi-specialized VIEW-AS-INTERFACE methods, however, allow extension to other languages and thus to classes of recipients other than browser-oriented humans.
We expect to create a KQML- or FIPA ACL (Foundation for Intelligent Physical Agents Agent Communication Language [FIPA])-based mechanism similar to the existing HTML method to enable agent-to-agent communication. This task would consist in creating a message for each message class of interest but specialized on the new interface class. The task should proceed independently of the agents’ reasoning mechanism because the types of messages and their semantic content shouldn’t change.

**Preservation of state in a stateless environment**

A stateless process is one whose state never differs from its startup state; a process whose state can differ from its startup state is stateful. Any display process is thus stateful, because the display changes. Such state changes, however, do not affect the behavior of the browser mechanism. For the purposes of this paper, we distinguish state changes that can affect the behavior of the stateful mechanism as affective and those that do not as nonaffective. Changing the channel on a television set is affective; sending a different set of scan lines to its display in nonaffective. When mechanism “A” is unable to cause a behavior change in mechanism “B”, A is nonaffective with respect to B.

Page servers are nonaffective with respect to Web browsers, by design, because the behavior of the browser is totally independent of page content. No state of the server is allowed to result in anything more or less than a displayable page of text, even unrecoverable errors.

Server nonaffectiveness raises some significant issues in using a browser as a client. One is that the display of the browser is updated only at the request of the user; this is the requested-update-only problem. Another is evident in browsers that retain multiple active display states (i.e., buttons on every page, including “back” and “forward” are continuously operational). Together, these mean that not only can the client display show incorrect information about the server state, but also that the server can get requests from the client that reflect neither its own state nor its model of the client state. In short, it’s very easy for the server and the client to get de-synchronized, especially when the user enters the act. Together these result in a severe integrity problem: the browser commands can readily cause damage to server data structures if they’re inappropriate to what data are actually present.

This difficulty doesn’t arise in the nominal client-server model; on the client side, the display is slaved directly to the client, so that when the client state changes, so does the display. The client behaves just like any other standalone program. Furthermore, the client is nonaffective with respect to the server, which merely executes requests with no knowledge of client state (excepting, of course, what is embedded in the requests). Any needed state information must be retained in the client. As an aside, this is entirely satisfactory when the server is answering database queries, because all query information can be retained in the client and transmitted in the request.

This model cannot be used when the server is constructive, that is, when the server builds a data structure for the user. The server must retain state of what the user is doing; the state is, in general, too complex to include in every user/server transmission as a parameter. In standalone applications, this is not an issue because the display is slaved to the computing mechanism. In remote applications, the server can “push” information to the client when the state changes.

“Server push” is becoming more visible on the Web: Servers “push” pages to the client that were not specifically asked for. It is not under control of the client. Server push can be useful in, e.g., continuous update situations. You essentially get several pages for the price of one. It is also not currently possible for a server to push a page to a client unless an initial request has been made. This enables pushed pages to go right through any proxy barriers because they have the required “requester” tag. In particular, a server can’t send a page to a client whenever it feels like it. This would be undesirable because it allows the server to gain control of the client, which is inherently dangerous. The client can abort the download, but “getting more than you asked for” is contrary to
the prominent client model.

In essence, then, the problem is that a nonaffective server (by extension, all Web servers), even if it retains state and gets control of the browser, cannot—and should not be able to—affect the state of the browser. The browser, in turn, cannot retain information about the computational state to enable the usual client/server model, which assumes that state information resides in the client. Therefore, building large, complex data structures for storage, retrieval, and later use seems to be impossible.

A designer wishing to use a stateless display as an interface into a stateful process must utilize a mechanism that: (a) generates display instructions reflecting the process state (without loss of generality, meaningful interaction with a stateful process implies that its state be observable) (b) records the process state when display information goes out to the user, (c) preserves the state until a response is received, (d) when the response is received, revives the appropriate process state, (e) updates the process state as needed based on information in the response.

In a synchronous environment, b, c, and d are trivial, because the process simply waits while the user responds to the display, which is to say, the user and the computational process act as lock-step co-routines. The problem becomes more complex in the non-synchronous multi-user environment because each step must be dealt with explicitly.

CHI provides (a) - (f) transparently to the interface designer. CHI does this by creating, at display time, a unique instance that represents the page that the user is to see. CHI then sends sufficient information to uniquely identify the instance to the user's browser, along with everything the user sees. At response time (e.g., when the user submits a form that's on the page), that unique ID is returned to the CHI Web server as part of the standard HTTP [Berners-Lee] packet of information (both “POST” and “GET” support this, and CHI deals correctly with both).

The receiving ACGI function dissects the HTTP message and translates the contents into CHI terms. Among the received items is the unique identifier of the page instance from which the user's browser page was generated. Since this page instance is unique, and can be accessed only by its unique ID via this mechanism, the page instance not only preserves any relevant state information, it also provides the uniquely retrievable reconnect point for responses from this particular user to this particular state. From this page instance, all aspects of the process state that pertain to that user are available and no aspects that don’t pertain to the user are available.

Case study: an instance of CHI in use

In 1997 the Advanced Information Systems Laboratory (AISL) at Sandia National Laboratories completed a prototype of the Border Trade Facilitation System (BTFS) [Goldsmith et al.]. The BTFS addresses three issues that can cause severe time delays in the border-crossing process: (1) manual entry of redundant information by different organizations; (2) incomplete regulatory documents; and (3) lack of timely shipment status information. The BTFS comprises multiple autonomous software agents that assist human actors by creating, documenting, monitoring and coordinating shipping transactions. Although it is merely a prototype, the BTFS permits international commerce operations from any site equipped with an Internet connection and a web browser. In this section of the paper we comment on utilization of CHI (and HCHI) in the process of developing a large, complex, distributed ecommerce application.

The BTFS design is based on three general concepts:

1. The BTFS is constructed within a secure distributed object framework, developed at the AISL. We have addressed certain security issues involving distributed access, the discussion of which we leave for another forum. Details of the underlying distributed object infrastructure are available in [Spires].
2. Shipping is controlled by software agents. The AISL has developed the Standard Agent Framework (SAF) to enable the construction of agents to carry out distributed tasks.

3. A shipment is controlled and documented via a composite object distributed among the relevant agencies. This object is called the Maquiladora Enterprise Transaction (MET).

BTFS agents interact with the border-crossing process by modifying the MET (agents can contact humans and other agents directly, as well). Control is entirely data-driven; agents modify the MET based on their goals, their knowledge and the state of the MET. Changes in the MET can trigger agent behavior in a manner reminiscent of a blackboard system [Englemore and Morgan].

As a system, the BTFS communicates with humans via the Web. Dozens of forms, each with tens to hundreds of data elements, are required. One such form, the Mexican customs Pedimento, requires 52 separate data entries to ship an item (each additional item requires 14 additional data entries). We leave aside the problem of interrelating the information needed by several stakeholders, including two different national governments, in this paper (but see [Phillips et al.]). What remains is the construction of the actual forms to be used and the weaving together of all the data on those forms with a distributed application and database. This is a relatively large and painstaking task, but becomes particularly difficult against the maintenance requirement to support changes in both the forms themselves and in the underlying data structure (based on, say, new information from stakeholders).

CHI and HCHI in essence trivialize this task. The process of building and entering a new form into the BTFS is extremely simple:

1. Design the web page, taking care to assign the correct names to the data entry points;
2. Compile the page into its class definition and initialization function using HCHI;
3. Insert the MAKE-OBJECT method call into the appropriate ENSURE-NEXT-PAGE-OBJECT definition so that the page gets built at the proper time.
4. Write code in the ENSURE-NEXT-PAGE-OBJECT definition using the named variables from the page as needed (utility functions can supply them as needed). Correspondence between variables in the page object and other variables in the application can be intricate but is normally straightforward.

Several dozen complex forms were built in two person-weeks, and this relatively small time expenditure did not affect the progress of the application itself. Maintenance of these pages continued through the end of the project, with pages often being “compiled” by HCHI several times a week. The bottom line: The interface design for this extremely complex application was straightforward, easy to maintain, and functioned properly throughout the demonstration and delivery period.

Conclusions

The CHI mechanism is an effective, efficient, and extensible means of agent/human communication. State can be preserved on the server based on unique CHI identifiers present in messages to and from the client, thus effectively rendering not only the client but also the entire interface process transparent to the application designer. This abstracts the application design entirely from the interface, which means the interface and the application can be designed asynchronously as long as the terms of correspondence are named in advance. Because the application is so narrowly coupled to the interface, new interface paradigms and languages can be readily substituted. The interface is similarly free and can be modified and updated for new languages and facilities. The same application can serve several interface types due to the ability to specialize methods on multiple arguments. The dynamic capabilities of CLOS also permit classes to be defined at runtime, offering the possibility of interfaces designed dynamically based on user input and discovered information.
Bibliography


[FIPA] Foundation for Intelligent Physical Agents Website <http://drogo.cselt.stet.it/fipa/>


