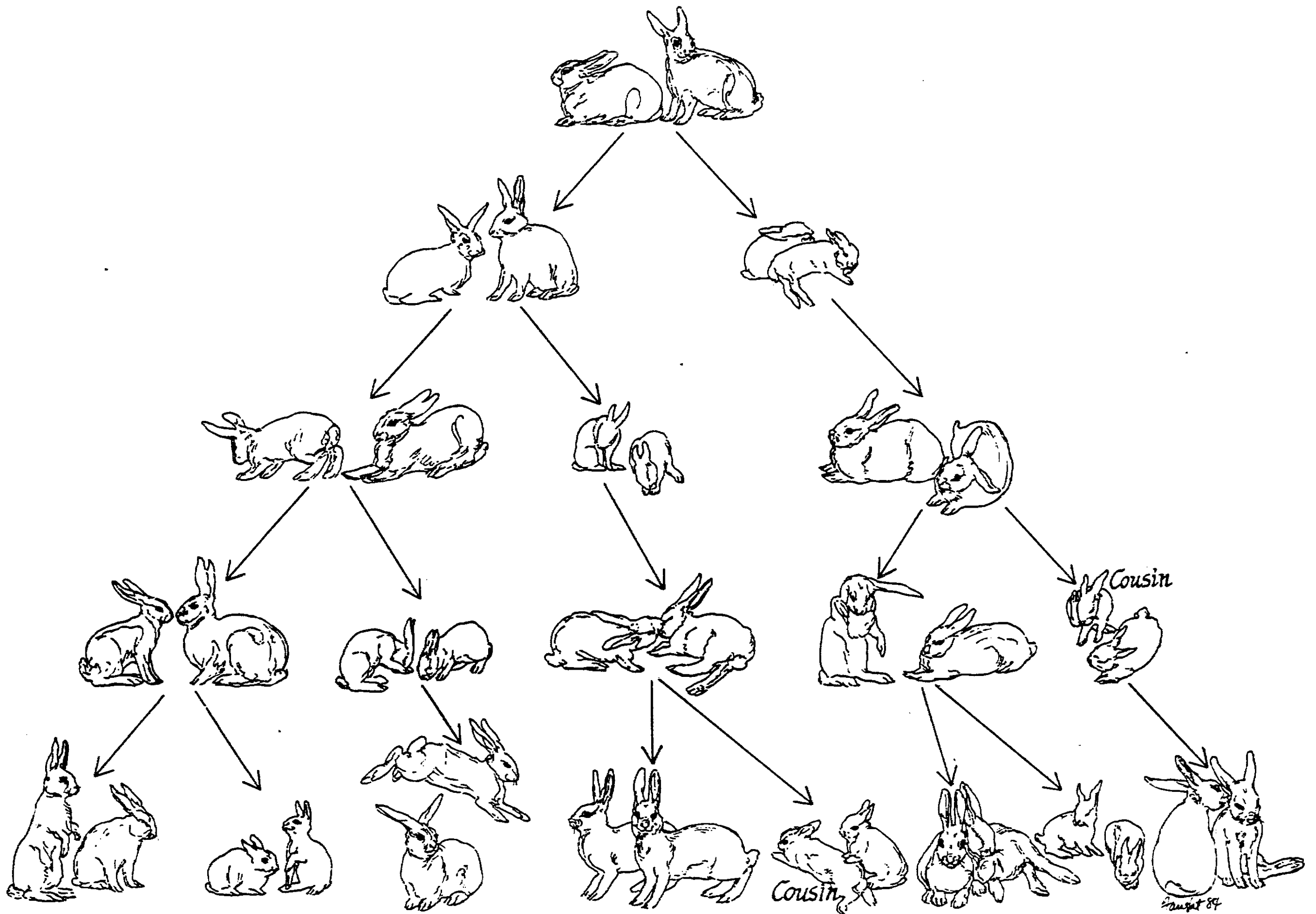
CARNEGIE-MELLON UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE

SPICE PROJECT

Cousin Application Builder's Manual

Phil Hayes, Rick Lerner, and Pedro Szekely



It is easier with a Cousin

23 August 1984

Spice Document S158

Location of machine-readable file: [cad]/usr/cousin/doc/applbldr.press

Copyright © 1984 Carnegie-Mellon University

This is an internal working document of the Computer Science Department, Carnegie-Mellon University, Schenley Park, Pittsburgh, Pennsylvania 15213 USA . Some of the ideas expressed in this document may be only partially developed, or may be erroneous. Distribution of this document outside the immediate working community is discouraged; publication of this document is forbidden.

Supported by the Defense Advanced Research Projects Agency, Department of Defense, ARPA Order 3597, monitored by the Air Force Avionics Laboratory under contract F33615-81-K-1539. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Projects Agency or the U.S. Government.

Table of Contents

1. Introduction 1
2. Goals of COUSIN Project 1
   2.1. Cooperative interfaces 2
   2.2. Application-independent interfaces 3
3. Form-Based Communication 3
4. Application Descriptions 6
   4.1. Example application description 9
5. COUSIN Application Interface 15
   5.1. Naming 15
   5.2. Routines to Access and Update Values in the Form 15
   5.3. Control functions 24
   5.4. Example Application Program 25
6. Building a COUSIN Application - A Summary 28

List of Figures

Figure 1: Communication by Form-Filling 4
Figure 2: Form for cmuftp 14

1. Introduction

COUSIN is a system that provides cooperative, form-based, command interfaces for a variety of application programs. The version of COUSIN described here runs on the Perq under Accent and provides graphically-oriented user interfaces for application programs running under Accent. COUSIN communicates with the application programs via the Accent Interprocess Communication facility (IPC) and with the user via the Sapphire window management system. The exact form of the interface seen by the user for a particular application is defined by a declarative *application description* file, which is interpreted by COUSIN. To employ COUSIN, an application builder needs only to construct this file for his application and to include the appropriate IPC calls in his program. This document explains the format of the application description file (Section 4), the kinds of messages that pass between COUSIN and an application system, and the Pascal[1] functions that are provided to facilitate that message passing for the application builder (Section 5). A quick summary of all the steps necessary to create an application that uses COUSIN for its user interface is contained in Section 6. Sections 4, 5, and 6 reflect the current implementation of COUSIN under Accent. This implementation is under active development, and some of the details given may be subject to change. Preceding these manual sections, Sections 2 and 3 give some more general background on COUSIN (see [3, 2] for further details). While this document explains how to build a COUSIN interface to an application system, it does not give full details on how the end user can interact with such an interface to control the application program. For this information, the reader should consult the Cousin User's Manual, the reference manual for end users of applications with COUSIN interfaces, available in [cfs]/usr/spice/cousin/dev/cousin.press.

2. Goals of COUSIN Project

The COUSIN (*Co*operative *Us*er *In*terface) project is a research effort aimed at producing better user interfaces. Currently, we are working only on coarse-grained command interfaces (e.g. interfaces to electronic mail systems or operating systems, but not text editors or drawing systems). Our approach is to produce interfaces that are not only *cooperative* (i.e. user-friendly) but also *application-independent* (i.e. the same monolithic interface can be used with a wide variety of application systems.) There are two implementations of COUSIN, one for the Perq under Accent (see Sections 4 and 5 for details), and the other for the Vax under Unix[2] (see [3] for details). The background material in this and the next section is not specific to either of these implementations, but

---

[1] Analogous functions will be provided eventually for other languages.

[2] The Unix version is no longer supported

serves to explain the broad principles underlying both implementations.

2.1. Cooperative interfaces

We consider the following kinds of user-friendliness or cooperativeness to be crucial for command interaction:

- **Error handling:**
  - **error detection and reporting**: Detection of errors in the specification of commands and/or their parameters, and informing the user what the command or parameter should be.
  - **error correction**: Where possible, correction of erroneous commands or parameters to valid values. The user should always be given the chance to refuse the correction.
  - **error negotiation**: Negotiation with the user about commands or individual parameters when correction is not possible or results in several possible alternative corrections. This negotiation can involve presenting the user with alternatives (either corrections or from a universe of all possible values), giving the user more general information about what is required, or allowing the user to execute any other commands. Negotiating about one parameter need not involve other parameters to the same command that have already been completely specified.
- **Explanation facilities:**
  - **constantly available**: the user should be able to ask for help at any time.
  - **context sensitive**: the kinds of help offered to the user should be sensitive to what he is doing at the time, and in particular, to what problems he has run into.
  - **chunked in small pieces**: as opposed to coming in, say, manual entries of several pages in length.
  - **highly interconnected**: so that a user can easily find material semantically related to the material he is currently looking at.
- **Personalization:**
  - **vocabulary:**
  - **screen layout:**
  - ...

2.2. Application-independent interfaces

COUSIN interfaces are designed to provide interface services to a wide variety of application systems, and so contain no knowledge of any individual application, obtaining their information instead from a declarative *application description* file that the application builder must provide. There are several reasons for adopting this approach:

- **Implementation effort**: Building a highly cooperative interface requires a lot of implementation effort, and such effort cannot be justified for each of a large number of different application systems. The more sophisticated the interface hardware, the more pressing the need to share interface effort becomes. Cooperative interfaces require much more effort to construct for bit-map workstation personal computers than for terminal-based systems.

- **Consistency**: Monolithic interfaces provide consistency across all applications that they service. Again, this is much more important for workstations than for terminals (e.g. mouse button usage).

- **Easy incorporation and testing of new interface features**: It is only necessary to change one program to implement and evaluate a new interface feature on a system-wide basis.

3. Form-Based Communication

To meet the twin goals of cooperative interaction and application-independence, COUSIN interfaces employ a form-based model of communication. Each application has an associated form analogous to the kind of business form in which information is entered by filling in blanks, or circling alternatives. The fields of the form correspond to the various pieces of information that the user and application need to exchange during an interactive session including input parameters, output from the application, and subcommands to the application. The lower part of Figure 1 shows the form for a generic print application program. In this example, the fields all correspond to input parameters. Some fields have default values as indicated by the square brackets for 'Font' and 'PageHeadings'. Such defaults can be overwritten by the user on input fields as has happened in this form instance for 'Copies' and 'Recipient' (defaults 1 and "Self" respectively). Some input fields, 'Files' in this example, have no default and must be specified by the user. The application description mentioned earlier can be thought of as a prototype or blank form for its application, and contains additional information associated with each field, including its type, the way to present it to the user, and how the user can change it. The types can be of varying levels of specificity, ranging from *String* for 'Recipient' through *Integer* for 'Copies' and *ReadableFile* for 'Files' to enumerated types for 'Font' and 'PageHeadings', the latter being an enumeration of size two (Included, NotIncluded). Methods of presentation and ways to change can include menus and buttons (not shown in the example).

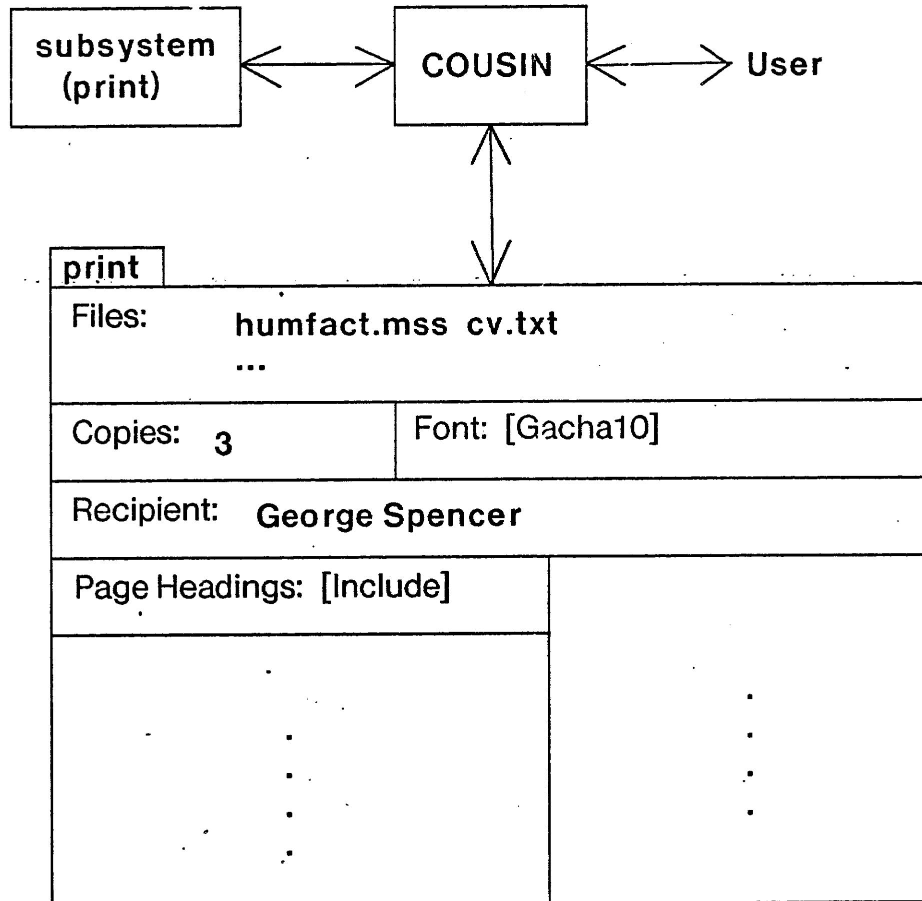


Figure 1: Communication by Form-Filling

As indicated by the upper part of Figure 1, a user and an application program communicate indirectly by reading and updating fields of the form for that application with all access to the form controlled by the COUSIN interface system. This effectively decouples the application system from direct interaction with the user, and allows COUSIN to be application-independent. The application need only specify via its form what information it wishes to have input and output, and COUSIN will manage the interaction that realizes that transfer of information to and from the user. The types and defaults associated with the form also provide the basis for cooperative interaction by allowing COUSIN to enforce the field types on input fields through error-correcting dialogues with the user. In addition, the information in the forms can be used to generate explanations about the application, its parameters, and subcommands in response to requests for help by the user (see [1] for more details).

The example forms shown above contains only input fields. This kind of form is suitable for the initial specification of the parameters of non-interactive applications, but is clearly less than sufficient for interactive applications. Nevertheless, form-based communication can be used with interactive

applications. There seem to be three general styles of communication with applications that can be supported straightforwardly through a form-based approach:

- **Non-interactive**: Parameters are specified. usually in a command line which is collected by a system command interpreter, before execution of the application begins. The application normally runs to completion after being invoked in this way.

- **Information collecting**: The application may accept (or request) additional information after it gets control, either because necessary parameters were omitted in the initial command line or because a need for additional information is discovered after execution begins.

- **Command loop**: After start-up, the application enters an interactive command loop: repeatedly accepting commands, executing them, and presenting the results to the user, who then composes his next request.

The corresponding scenarios for the form-based approach of COUSIN are:

- **Non-interactive**: This is the simplest case. Form fields correspond directly to application parameters. The user invokes the application through a menu or a command line, which may specify values for some (or all) of the input fields for the application. COUSIN obtains the form for the application thus specified, and sets up its defaults. If a command line was used, COUSIN parses it and transfers the various parameter values thus obtained to the appropriate fields of the form. If after this, all parameter fields are correctly filled, COUSIN executes the application in the normal way. If, on the other hand, information is missing or incorrectly specified, COUSIN reports the problems to the user, and gives him an opportunity to correct the situation by editing the form. When both the user and COUSIN are satisfied with the way all the fields are filled, the user may start execution of the application explicitly. However, COUSIN will not allow him to start execution while problems remain with the form. If the user is unable to correct the form satisfactorily, he must either abort the attempt at application invocation or save the form in its current state for later correction.

- **Information collecting**: This situation is similar to the previous one, except that COUSIN will start execution of a application with some of the required parameter fields unspecified. After the application is started, it can request the value of any field in its form. If a requested field is undefined, COUSIN will inform the user that a value is required and suspend execution of the application until the user specifies the required value which is then checked and, if correct, passed back to the application. Using this type of interaction, an application can be started without fillers for any of the fields in its form being specified, and the user interface will prompt for whatever parameters are needed when they are first referenced. It is a good example of how COUSIN insulates the application from concerns about how and in what order its parameters are acquired, and yet can make the parameters available as they are required.

- **Command loop**: The user specifies interactive commands to the application by inserting the name of a command into a field whose type is an enumeration of all the commands available; this insertion can be done by direct type in or by menu selection. Alternatively, there can be a "button"-valued field for each possible command. In either case, the field used to communicate the command has a special *active* status which means that a

message is sent to the application by COUSIN every time the field changes value, thus allowing the application to avoid inefficient polling of the field's value.

When not actually executing one of its own commands, the application would wait for notification that one of these *active* fields had been modified. Additional parameters for application commands can be specified through other fields in the form in the same way as the two previous cases, and information about which fields serve as parameters to which subcommands can be included with the form. Using this information, when the user issues a command, COUSIN can indicate to the user which fields provide parameters, and make sure that these fields contain correct values, not informing the application of the command unless they do. Facilities are also needed to allow the application to determine whether such parameter fields are up to date or are merely an inappropriate leftover from earlier invocations of subcommands.

In each of the above cases, results can be transmitted back from the application to the user as the values of additional fields reserved for that purpose, and modifiable only by the application. COUSIN will display these field values to the user.

4. Application Descriptions

After the general background in the two preceding sections, this and the following section contain material specific to the current implementation of COUSIN for the Perq under Accent. This implementation is under active development and some specific details given here may be subject to change.

As already described, in order to use COUSIN interface facilities, applications must have an application description which specifies the form that COUSIN will use for communication between the user and that application. This section explains the format of application descriptions for the Perq/Accent version of COUSIN. Forms in COUSIN are composed of fields. Fields are just containers for values, and they have a set of properties which describe the kind and number of values that can be put in them, how those values are to be presented to the user, and how the user can modify the values. The kind of values supported by COUSIN are integers, strings, booleans, buttons, ports, and forms; forms which are the values of fields in another form are called subforms.

Application descriptions are text descriptions of forms. We are planning a comprehensive special purpose editor to create and edit these system descriptions. Currently, editing support exists only for the layout attributes of the descriptions, but they can also be created with a regular text editor. The format of application descriptions is a header which contains the name of the form:

```
[
StructureType: FormHeader
FormName: String
FormWidth: Integer
FormHeight: Integer
]
```

plus a description in the following format for each of the fields of the form:

```
[
StructureType: Field
Name: String
ExternalName: String
NamePosition: {Top, Left, NotPresent}
ValueType: {Integer, Boolean, String, Button, SubForm, Port}
MaxNumber: Integer
MinNumber: Integer
EnumeratedValue: (value1, value2, ...)
LowerBound: Integer
UpperBound: Integer
DefaultSource: {NoDefault, ExplicitDefault, ApplicationDefault, UseSubfieldDefault}
DefaultValue: String
ChangeResponse: {Passive, InformApplication, Command}
Parameters: (FieldName1, FieldName2, ...)
InteractionMode: {Table, EditIn, PushButton, CycleButton,
        DisplayOnly, Invisible, Typescript, Canvas}
NumRows: Integer
NumColumns: Integer.
HasScrollBar: Boolean
DeselectedMark: {NoMark, Bold, Overstrike, LightBox, HeavyBox, Invert}
SelectedMark: {NoMark, Bold, Overstrike, LightBox, HeavyBox, Invert}
Tokenize: Boolean
Presence: Boolean
LeftX: Integer
RightX: Integer
TopY: Integer
BottomY: Integer
Switches: (SwitchDescription1, SwitchDescription2, ...)
]
```

Some of these fields require further explanation:

ExternalName: this is the string that is displayed as the field name. The "Name" field is used for communication between cousin and the application. If ExternalName is not specified then Name is used.

NamePosition: Specifies the position of the external name in the field (only if *InteractionMode* is *Table*).

MaxNumber, MinNumber: the minimum and maximum number of values permitted in the field (both default to 1).

EnumeratedValue: an initial list of possible values that the field is restricted to.

LowerBound, UpperBound: on the value if the field has an integer value.

DefaultSource:

NoDefault: the field has no default (the default).

ExplicitDefault: the default is listed explicitly in the application description.

ApplicationDefault: the field has a default which will be determined dynamically by the application, so the user can leave this field empty and yet it is still counted as having a correct value.

UseSubfieldDefault: the field has a ValueType of SubForm, and the default for the field is whatever subform is obtained by taking the defaults of all the fields of the subform.

ChangeResponse:

Passive: only check for validity when a new value is entered; do not tell the application (the default).

InformApplication: tell the application as well

Command: first check the values of the fields listed as parameters, and inform the application only if all these values are correct (see Section 5).

Parameters: the list of parameters for use when ChangeResponse is Command.

InteractionMode: the way the field is presented to the user and the way he is allowed to interact with it.

Table: displays a universe of values for the field in a tabular format. The user can then select and deselect individual values from the universe with the pointing device. It is a kind of menu.

EditIn: the user types the value into the field (the default).

PushButton: the field appears like a button to be pushed and has two values (ButtonOn and ButtonOff); requires ValueType to be Button.

CycleButton: the field name does not appear, only the value, and repeatedly selecting the field causes the value to cycle through the range of possibilities.

DisplayOnly: for output from the application.

Invisible: the value is not displayed (for use with passwords).

Typescript: for typescript interaction between the application and user directly, or for scrolled output from the application.

Canvas: for any kind of output or direct user interaction that the application wishes.

NumRows, NumColumns: This applies only if *InteractionMode* is *Table*. These fields set the number of rows and columns in the table. If NumRows is set to 0 Cousin will pack the rows as close as possible.

HasScrollBar: If *InteractionMode* is *Table*, this specifies whether or not the table has a scroll bar.

SelectedMark, DeselectedMark: If *InteractionMode* is *Table*, these specify how the selected and deselected entries are marked.

Tokenize: If set to true spaces will be interpreted as value separators within a field. Defaults to true.

Presence: If set to false the field will not be displayed. Defaults to true.

LeftX, RightX, TopY, BottomY: specify the coordinates of the field in the form. The units are pixels relative to the top left corner of the form.

Switches: are used to specify values for fields from the command line. Switches are used to indicate how to associate tokens in the command line with fields in the form. There are two kinds of switches:

Positional. A positional switch indicates that the field is filled from the token in a specified position in the command line. The syntax for positional switches is **//Position***k* where *k* can be 1, 2, 3, ... or **//PositionAll**. For example, //Position2 appearing as the switch for a field means that the second token in the command line should go into that field. //PositionAll means that all tokens in the command line should go into that field.

Named switches: the named switches define keywords that can be used in the command line to refer to a field. If a switch doesn't have a value associated with it, the next token in the input is interpreted as the value. This kind of switches are specified just by giving a list of names that can be used in the command line. Switches can have values associated with them, so when the switch name appears in the command line the corresponding value is given to the field. Input tokens accounted for by named switches are ignored from the point of view of positional switches. The example application description that follows illustrates the use of switches.

4.1. Example application description

To make this definition more concrete, consider the following application description which would allow COUSIN to provide a graphical form interface for cmuftp, a program widely used in CMU CSD for transferring files across the local area network.

```
[
StructureType: FormHeader
FormName: cmuftp
]

[
StructureType: Field
Name: "Files To Transfer"
ValueType: String[3]
MinNumber: 1
MaxNumber: 50[4]
]

[
StructureType: Field
Name: Send
ValueType: Button
InteractionMode: PushButton
ChangeResponse: Command
Parameters: ("Files to Transfer" Host Mode Account Password
             "Foreign Prefix" "Foreign Suffix" "Local Prefix" "Local Suffix")
]

[
StructureType: Field
Name: Receive
ValueType: Button
InteractionMode: PushButton
ChangeResponse: Command
Parameters: ("Files to Transfer" Host Mode Account Password
             "Foreign Prefix" "Foreign Suffix" "Local Prefix" "Local Suffix")
]
[
StructureType: Field
Name: Quit
ValueType: Button
InteractionMode: PushButton
ChangeResponse: Command
]
```

[3] File types like ReadableFile and WritableFile are not implemented yet, though they are planned

[4] 50 is an implementation restriction

```
[
StructureType: Field
Name: Host
ValueType: String
EnumeratedValues: (
(cad e)                        ;; a list in an EnumeratedValues
(spice x)                      ;; indicates several names for the
(zog z)                        ;; same member of the enumeration,
(vlsi v)                       ;; the first of which is the
(cmua a)                       ;; canonical name
)
DefaultValue: cad
Switches: (
                               ;; the switch "Host" can be used in the
                               ;; command line to indicate that the next
                               ;; token should be parsed as a value of the
                               ;; host field. The switch "EVax" also defines
                               ;; a value. When used in the command line it
                               ;; indicates that the Host field should be
                               ;; set to cad

  Host
  [ Name: EVax
    Value: cad
    ]
  )
]

[
StructureType: Field
Name: "Host Number"
ValueType: String
InteractionMode: DisplayOnly
DefaultSource: ApplicationDefault
]
```

```
[
StructureType: Field
Name: Mode
ValueType: String
EnumeratedValues: (
auto
binary
text
)
DefaultValue: auto
]

[
StructureType: Field
Name: Account
ValueType: String
DefaultValue: ""
]

[
StructureType: Field
Name: Password
ValueType: String
InteractionMode: Invisible
DefaultValue: ""
]

[
StructureType: Field
Name: "Foreign Prefix"
ValueType: String
DefaultValue: ""
]

[
StructureType: Field
Name: "Foreign Suffix"
ValueType: String
DefaultValue: ""
]

[
StructureType: Field
Name: "Local Prefix"
ValueType: String
DefaultValue: ""
]

[
StructureType: Field
Name: "Local Suffix"
ValueType: String
DefaultValue: ""
]
```

```
[
StructureType: Field
Name: Confirm
ValueType: String
InteractionMode: RegularMenu
EnumeratedValues: (
Disabled
Enabled
)
DefaultValue: Disabled
]

[
StructureType: Field
Name: Transfer Progress
ValueType: String
InteractionMode: Typescript
]
```

Since this description contains no formatting information, the form that COUSIN would produce from it would be formatted automatically. The formatting algorithm is straightforward, and packs as many fields as fit per line in the order specified in the description. A form that could be produced through insertion of formatting attributes (LeftX, RightX, TopY, BottomY, and FormWidth and FormHeight), is shown in Figure 2. The only difference is in the layout; the individual fields would look the same. There is currently a form layout editor to assist with the layout task. We have plans to expand this to a full interactive form design editor through which all aspects of a form could be specified. Such an editor could be used for personalization of forms by the end user, as well as for initial form design by the application builder. To improve efficiency, COUSIN does not interpet the text form description directly, but uses a preprocessed binary version. The utility program **compsf** converts the text format into binary format.

Figure 2 shows the form as it would appear to the user when it first started up: 'send' and 'receive' are command buttons; 'mode' and 'confirm' are displayed as explicit menus with the current selection shaded, selecting one of the other elements would cause the shading to move; 'Transfer Progress' is a scrolled typescript field in which the application can write messages to the user; 'Host Number' is an output only field that the user cannot type in; 'Password' is a no-echo field; all other fields are simple type in fields for the user. Some of the fields have initial defaults as specified in the application description.

In addition to maintaining the form on the screen and allowing the user to edit it, COUSIN also enforces the restrictions on field values, requiring 'Host', for instance, to be filled by a string that is a

cmuftp

send receive

File to transfer:

Host: cad | Host Number:

Mode: Auto Binary Text

Account: | Password:

Foreign Prefix: | Foreign Suffix:

Local Prefix: | Local Suffix: .

Confirm: Enabled Disabled

Transfer Progress:

Figure 2: Form for cmuftp

member of its enumeration. The user can put incorrect values in the fields, but COUSIN will complain, and offer corrections if it can. When a command button is pressed, COUSIN will also check that all the fields listed as parameters to that command are filled in correctly and will not transmit the command button push to the application until they are all correct.

5. Cousin Application Interface

In addition to providing a cooperative interface for the user, Cousin must also provide an interface to the application system. On the perq under Accent, this interface is provided through Matchmaker (the details of Matchmaker are described in the Spice Programmer's Manual). The following is description of the facilities that are implemented and of how to use them. All these facilities can be accessed by importing module **CousinCalls** from file **cousincalls.pas.**

5.1. Naming

In communicating with Cousin, an application must be able to refer to and understand references to three different kinds of object: forms, the fields in the form, and the values in the fields. Each of these objects is referred to by an *Id* of appropriate type.

Forms are referred to by a *FormId*, which is a port assigned to the forms by cousin. The FormId of the top-level form for an application is the global variable **TopForm** which is exported by the cousinuser module.

The next level of naming is the fields. Fields have ids, called *StaticIds*, relative to the form in which they are defined. The ids of the fields are automatically calculated from the application description file by a utility program called **genconst**, which takes an application description file and generates a Pascal source file which is suitable to be imported by the application program and which sets Pascal constants with the same symbolic name as the name of the field to the corresponding StaticId. Care should be taken to run **genconst** and recompile the application program when fields are added, deleted or reorganized in the sdo (application description) file. The second way to name fields from applications using full symbolic names.[5] Application builders are encouraged to use the ids because they are easier to use and more efficient.

The values in the fields are viewed as a list of values, and the application can refer to them by their position in this list.

5.2. Routines to Access and Update Values in the Form

There are numerous routines to access and update values in forms. The routines are in module **CousinUser** which is automatically imported by **CousinCalls**. The following is a description of each of the routines available[6]. This description is in the form of an annotated copy of the Matchmaker

---

[5]The symbolic names are rather clumsy and might be changed in the future so they are not documented here.

[6]For the latest description see file cousin.defs from which which the actual calls are generated.

interface definition file.

```
{
{ routines to get the value of a field
{
{ CForm         The form to get the value from
{ fid           The static id of the field within the CForm form
{ index         Get the index-th value of the field. If the field only has
{                       one value use 1.
{ val           The value returned
{
{ Returns       Success or Failure; If return is failure the value in "val"
{               is undefined. The most likely reason for that is that
{               the parameters are incorrect.
{ }
routine GetIntField(    : CForm;
                        fid: StaticId;
                        index: Integer;
                    out val: Integer
                        );


routine GetBoolField(   : CForm;
                        fid: StaticId;
                    out val: Boolean
                        );


routine GetStrField(    : CForm;
                        fid: StaticId;
                        index: Integer;
                    out val: EString
                        );



routine GetSubFormField(        : CForm;
                        fid: StaticId;
                        index: Integer;
                    out val: CForm
                        );



routine GetButField(    : CForm;
                        fid: StaticId;
                    out val: ButtonValue
                        );


routine GetPortField(        : CForm;
                        fid: StaticId;
                        index: Integer;
```

```
                    out val: Port
                        );

{
{ Not Implemented in the current version
{ }
routine GetContextForm(            : CForm;
                    out form: CForm;
                    out fid: StaticId;
                    out index: Integer
                        );

{
{ Given the index-th value in the value list find out its index in the
{ universe list
{
{ CForm          The form to get the value from
{ fid            The static id of the field within the CForm form
{ index          The index in the value list
{ uIndex         The index in the universe
{
{ Returns        Success or Failure; If return is failure the value in "uIndex"
{                is undefined. The most likely reason for that is that
{                the parameters are incorrect.
{ }
routine GetUnivIndexField(           : CForm;
                        fid: StaticId;
                        index: Integer;
                    out uIndex: Integer
                        );


{
{ Get the indices in the universe of all the values in the field
{
{ CForm          The form to get the value from
{ fid            The static id of the field within the CForm form
{ seq            Will contain the indices. The storage for the array will
{                        be allocated by COUSIN, so be careful to deallocate it.
{
{ Returns        Success or Failure; If return is failure the value in "seq"
{                is undefined. The most likely reason for that is that
{                the parameters are incorrect.
{ }
routine.GetAllIndexField(           : CForm;
                        fid: StaticId;
                    out seq: pIntegerArray
                        );


{
{ routines to add a value to a field
{ CForm          The form to get the value from
{ fid            The static id of the field within the CForm form
{ index          The place in the value list where the value should be added
{                        0  don't care
{                       -1  at the end
{ val            The value to add
```

```
{ }
SimpleProcedure AddIntField(    : CForm;
                        fid: StaticId;
                        index: Integer;
                        val: Integer
                        );


SimpleProcedure AddStrField(    : CForm;
                        fid: StaticId;
                        index: Integer;
                        val: EString
                        );


Procedure AddSubFormField(        : CForm;
                        fid: StaticId;
                        index: Integer;
                    out val: Integer;
                    out strRep: EString
                        );


{
{ routines to replace a value in a field. If the field is empty the value
{ is added
{
{ CForm          The form to get the value from
{ fid            The static id of the field within the CForm form
{ index          The index of the value to replace
{ val            The value to add
{ }
SimpleProcedure RepIntField(    : CForm;
                        fid: StaticId;
                        index: Integer;
                        Val: Integer
                        );


SimpleProcedure RepBoolField(   : CForm;
                        fid: StaticId;
                        Val: Boolean
                        );


SimpleProcedure RepStrField(    : CForm;
                        fid: StaticId;
                        index: Integer;
                        Val: EString
                        );
```

```
procedure RepSubFormField(       : CForm;
                        fid: StaticId;
                        index: Integer;
                    out Val: Integer;
                    out StrRep: EString
                        );


SimpleProcedure RepButField(    : CForm;
                        fid: StaticId;
                        Val: ButtonValue
                        );


{
{ Delete a value from a field. The element is not deleted form the universe
{
{ CForm          The form to get the value from
{ fid            The static id of the field within the CForm form
{ index          The index of the value to delete
{ }
SimpleProcedure DelField(       : CForm;
                        fid: StaticId;
                        index: Integer
                        );


{
{ Delete all the values of the field. The universe is not deleted
{
{ CForm          The form to get the value from
{ fid            The static id of the field within the CForm form
{ }
SimpleProcedure WipeField(      : CForm;
                        fid: StaticId
                        );


{
{ Set the value of a field to the default if there is one
{
{ CForm          The form to get the value from
{ fid            The static id of the field within the CForm form
{ }
SimpleProcedure SetDefaultField(        : CForm;
                        fid: StaticId
                        );


{
{ Routines to add values to universes. Adding elements to universes only is
{ MUCH faster than adding values (to value lists). The values are NOT checked.
{
{ CForm          The form to get the value from
{ fid            The static id of the field within the CForm form
```

```
{ index         The place in the universe where the value should be added
{                       0  don't care
{                      -1  at the end
{ val           The value to add
{ }
SimpleProcedure AddIntUniverse(        : CForm;
                        fid: StaticId;
                        index: Integer;
                        Val: Integer
                        );



SimpleProcedure AddStrUniverse(        : CForm;
                        fid: StaticId;
                        index: Integer;
                        Val: EString
                        );

{
{ Add a sequence of string values into the universe. The result is the same
{ as repeatedly using AddStrUniverse, but is a LOT faster.
{ Adds strings Val↑[first] to Val↑[Last] instead of a single value.
{ }
SimpleProcedure AddStrSeqUniverse(     : CForm;
                        Sid: StaticId;
                        index: Integer;
                        Val: pEStringArray;
                        first: Integer;
                        last: Integer
                        );


procedure AddSubFormUniverse(     : CForm;
                        fid: StaticId;
                        index: Integer;
                    out Val: Integer;
                    out strRep: EString
                        );

{
{ Mark an element of the universe as selected or deselected. Selecting a
{ value means that the value is inserted to the value list. Deselecting means
{ that the value is deleted from the value list (not from the universe)
{
{ CForm         The form to get the value from
{ fid           The static id of the field within the CForm form
{ index         The in the universe of the value to select/deselect
{ val           True ==> Select; False ==> Deselect
{ }
SimpleProcedure SelectElementUniverse( : CForm;
                        fid: StaticId;
                        index: Integer;
                        Val: Boolean
                        );
```

```
{
{ Select or deselect all the members of the universe
{
{ CForm          The form to get the value from
{ fid            The static id of the field within the CForm form
{ val            True ==> Select; False ==> Deselect
{ }
SimpleProcedure SelectAllUniverse( : CForm;
                        fid: StaticId;
                        Val: Boolean
                        );


{
{ Select or deselect the elements listed in the seq array.
{
{ CForm          The form to get the value from
{ fid            The static id of the field within the CForm form
{ seq            The array of indexes of elements of the universe
{                        to select/deselect
{                        only use the indices in seq↑[first] to seq↑[last]
{ val            True ==> Select; False ==> Deselect
{ }
SimpleProcedure SelectSeqUniverse( : CForm;
                        fid: StaticId;
                        seq: pIntegerArray;
                        first: Integer;
                        last: Integer;
                        Val: Boolean
                        );


{
{ Delete a value from a universe
{
{ CForm          The form to get the value from
{ fid            The static id of the field within the CForm form
{ index          the index in the universe of the value to delete
{ }
SimpleProcedure DelUniverse(    : CForm;
                        fid: StaticId;
                        index: Integer
                        );


{
{ Routines to delete values from universes. These functions are used to delete
{ a value by giving the value you want to delete instead of the index.
{ This is very inefficient.
{
{ CForm          The form to get the value from
{ fid            The static id of the field within the CForm form
{ val            The value to delete
{ }
```

```
SimpleProcedure DelIntUniverse(         : CForm;
                        fid: StaticId;
                        Val: Integer
                        );



SimpleProcedure DelStrUniverse(         : CForm;
                        fid: StaticId;
                        Val: EString
                        );



SimpleProcedure DelSubFormUniverse(     : CForm;
                        fid: StaticId;
                        Val: Integer
                        );

{
{ Delete all the elements of the universe
{
{ CForm          The form to get the value from
{ fid            The static id of the field within the CForm form
{ val            The value to delete
{ }
SimpleProcedure WipeUniverse(   : CForm;
                        fid: StaticId
                        );


{
{ Not implemented in this version
{ }
SimpleProcedure ShowAlternatives(       : CForm;
                        fid: StaticId
                        );



{
{ Get the name of a given form
{
{ CForm          The form to get the value from
{ val            The name of the form
{ }
procedure GetNameForm(    : CForm;
                    out val: EString
                        );


{
{ Set a field to be present or not in the displayed form
{
{ CForm          The form to get the value from
```

```
{ fid            The static id of the field within the CForm form
{ val            True ==> Make it present; False ==> Not Present
{ }
SimpleProcedure SetFieldPresence(      : CForm;
                    fid: StaticId;
                    Val: Boolean
                    );



{
{ Set the lock value of a field
{
{ CForm          The form to get the value from
{ fid            The static id of the field within the CForm form
{ val            True ==> Lock it; False ==> Unlock
{ }
SimpleProcedure RepFieldLock(   : CForm;
                    fid: StaticId;
                    Val: Boolean
                    );


{
{ Unlock everything; HAS to be called after commands are done
{ }
SimpleProcedure UnlockForm(    : CForm
                    );


{
{ Set the listener for a field; The activation messages for that field
{ will be sent to the given port.
{       Not fully implemented yet.
{
{ CForm          The form to get the value from
{ fid            The static id of the field within the CForm form
{ pt             The port where the messages should be sent
{ }
SimpleProcedure SetFieldServer(   : CForm;
                    fid: StaticId;
                    pt: Port
                    );

{
{ Set the field name that is printed on the display
{
{ CForm          The form to get the value from
{ fid            The static id of the field within the CForm form
{ name           The name that will appear on the display
{ }
SimpleProcedure SetFieldName(   : CForm;
                    fid: StaticId;
                    name: EString
                    );
```

```
{
{ Write a message in the pop up message windows
{
{ CForm         The form to get the value from
{ fid           The static id of the field within the CForm form; The message
{                      will appear close to this field
{ line          The message (only one line!!)
{ }
Simpleprocedure CousinMessage(  : CForm;
                       fid: StaticId;
                       line: EString
                       );

{
{ Exception will be raised when the request contains an error
{ }
Handler CousinError(  s: EString
                       );

{
{ Not implemented
{ }
Handler CousinAbort(  form: Port;
                       field: StaticId
                       );
```

5.3. Control functions

In the current implementation the control routines have been packaged in module **CousinCalls**. The routines in this module completely hide the complexity of the message passing from the application builder. In some cases fine control over the message passing is needed (for example if the application communicates via messages with another process besides cousin). This fine control is also available but will not be documented here; application builders who need it should contact one of the members of the Cousin group.

The following are the two routines defined in module cousincalls:

```
procedure InitCousinInterface;

procedure WaitForActivation(    var form      : CForm;
                                var field     : StaticId
                                    );
```

The routine **InitCousinInterface** should be called at the beginning of the program to establish communication with cousin. As a result of this call the variable **TopForm** will be set to the top level from associated with the application.

In addition to accessing and updating field values and universes, the application needs to be able to

find out from COUSIN when its subcommands are invoked and to obtain the parameters to those subcommands in a consistent state. This prevents, for instance, the user from changing the parameters between the time he issues the subcommand and the time COUSIN gets hold of the parameters. The current mechanism requires each subcommand to list (using the Parameters attribute of the field definition in the application description) the other fields that serve as parameters to it. When the user issues the subcommand, COUSIN checks the current state of all the fields named as parameters. If they are all correct, COUSIN *locks* them so that the user cannot change them, and informs the application of the command. The application is then supposed to retrieve the values it needs and issue an unlock command to COUSIN, whereupon all the fields become available for modification by the user as before. If not all of the parameter fields are correct, COUSIN does not inform the application that the user has issued the command, but engages in its usual error correcting behaviour on the incorrect fields. Once the fields are corrected, the user must reissue the subcommand; COUSIN will not automatically continue the process of issuing the command from the point at which the original attempt was blocked by the incorrect parameter fields.

The routine **WaitForActivation** is used to wait for activations from Cousin. When WaitForActivation is called it will wait until a message is sent to the application program. This routine will decode the message and return the *form* id and the *field* id of the field modified. WaitForActivation will receive messages for fields whose *ChangeRespose* is either *InformApplication* or *Command*, and it is up to the application program to know if the field corresponds to a sub-command or a parameter.

This locking mechanism is rather clumsy and facilities may eventually be added to allow a command and its parameters to be packaged and sent to the application as a single unit, eliminating the need for the application to request each value separately and then to unlock the form. The routine to do the unlocking is defined in module **CousinUser** (see UnLockForm in the previous section).

5.4. Example Application Program

This section illustrates how a Perq Pascal application program can be written to use COUSIN. The following program is the top level for the cmuftp application whose form was used as an example earlier in this document.

```
program cmuftp;

imports CousinCalls      from cousincalls;
                         {* the module that exports the functions
                         {* described above *}
imports cccmuftp from cccmuftp;
                         {* the file automatically generated from the
                         {* application description for cmuftp by the
                         {* genconst utility program to contain
                         {* definitions of constants whose names are
                         {* the field names and whose values are the
                         {* corresponding field ids *}

procedure DoSend; forward;

procedure DoReceive; forward;

{$include ccmuftp1.pas}          {* bodies for the procedures *}

var
  bool: Boolean;
  form: CForm;
  fieldId: StaticId;
  index: Integer;

begin
  InitCousinInterface;           {* initialize communication with COUSIN *}
  while true do
  begin
                         {* wait until a command is invoked *}
    WaitForActivation(form, fieldId);
                         {* find out which command was invoked *}
                         {* invoke the proper command handler *}
    case fieldId of
                         {* the constants are defined in file ccmuftp.pas *}
      CSend:    DoSend;
      CReceive: DoReceive;
      CQuit:    exit(cmuftp);
    end;
  end;
end.
```

Many simple COUSIN applications will follow this pattern: initialization, followed by a loop which waits for commands containing a case statement which dispatches on the command received. The procedures *DoSend* and *DoReceive*, which execute the commands do the real work. The following program fragment shows *DoReceive*:

```
procedure DoReceive;
var
  gr: GeneralReturn;
  hostName, mode, name: String;
  fPrefix, lPrefix, fSuffix, lSuffix: String;
  localFile, foreignFile: String;
  i: Integer;

begin
                        {* collect all the parameters that will be
                        {* the same for each file received *}

                        {* since the fields are parameters to the Receive
                        {* command we won't bother to check gr for success *}
{ host }
  gr := GetStrField(TopForm, CHostName, 1, hostName);
{ mode }
  gr := GetStrField(TopForm, CMode, 1, mode);
{ foreign prefix }
  gr := GetStrField(TopForm, CForeignPrefix, 1, fPrefix);
{ local prefix }
  gr := GetStrField(TopForm, CLocalPrefix, 1, lPrefix);
{ foreign suffix }
  gr := GetStrField(TopForm, CForeignSuffix, 1, fSuffix);
{ local suffix }
  gr := GetStrField(TopForm, CLocalSuffix, 1, lSuffix);

                        {* iteratively obtain the name of each
                        {* file to transfer and transfer it *}
  i := 1;
  cr := GetStrField(TopForm, CFilesToTransfer, i, name);
  while cr = Success do
  begin
    localFile := concat(lPrefix, name, lSuffix);
    foreignFile := concat(fPrefix, name, fSuffix);

                        {* SENDFILE does the actual sending *}
    SENDFILE(localFile, foreignFile, .....)

    i := i + 1;
    cr := GetStrField(TopForm, CFilesToTransfer, i, name);
  end;

  UnlockForm(form);     {* if all the file names were collected before
                        {* sends were done, the unlock could be done
                        {* in less time *}
end;
```

Again, this fragment illustrates a pattern that will be very common for COUSIN applications: collect parameters and execute command. In general, little or no parameter checking and interaction with the user in case of error will be required, because COUSIN ensures that the parameters satisfy the constraints specified in the form description, and the set of constraints supported by COUSIN cover

most common cases. Also, the application is unaware whether the parameters have been gathered by menu selection, type in, etc. or whether the values it receives have been spelling corrected. Because it relies on the services of COUSIN, its user interface code is trivial.

6. Building a COUSIN Application - A Summary

To create an application program called `myappl` that uses COUSIN for its user interface, proceed as follows:

- **Retrieve a Cousin system for applications builders.** The system for users doesn't include the files that are needed to compile the programs. An system appropriate for application builders can be retrieved via update using the logical name cousinbuildkit - a from the cfs vax.

- **Create the application description file**: according to the format described in Section 4. This file has to be created through an ordinary text editor. We are planning to build a specialized editor to facilitate the creation and modification of this files. Right now the best thing to do is to start from the description of another application. The resulting file should be called `myappl.SDO`.

- **Create a binary application description**: by running the utility program `compsf` with the command line `compsf myappl`. This will create a file called `myappl.BSD` containing a binary version of the application description file in the runtime format required by COUSIN.

- **Create the constants file**: which defines Pascal constants whose names are the names of the form fields and whose values are the corresponding field Ids. To do this run the utility program `genconst` with the command line `genconst myappl`. This will create a file, `cmyappl.Pas`, containing the constants, which you can `Import` into your own application program, `myappl.Pas`.

- **Write the application program**: using the Pascal calls described in Section 5 to implement the necessary communication between it and COUSIN. One example program is given in Section 5.4, others can be found in the `[cad]/usr/cousin/dev/......` update directories.

- **Compile and Link the Application program.** In order for the linker to find the necessary `.SEG` files, you must link your application with `cousin.RUN` (eg. link myappl, cousin).

Your application program can then be run with COUSIN providing its user interface as described in the COUSIN User's Manual for end users of COUSIN applications.

If you have any difficulties with the above procedure, please contact any of the authors of this document directly or by sending them mail at CMU-CS-CAD.

References

1. Hayes, P. J. Uniform Help Facilities for a Cooperative User Interface. Proc. National Computer Conference, AFIPS, Houston, June, 1982.

2. Hayes, P. J. Executable Interface Definitions Using Form-Based Interface Abstractions. In *Advances in Computer-Human Interaction*, H. R. Hartson, Ed.,Ablex, New Jersey, 1984.

3. Hayes, P. J. and Szekely, P. A. "Graceful interaction through the COUSIN command interface." *International Journal of Man-Machine Studies 19*, 3 (September 1983), 285-305.