1 The structure of this document

First a short overview of the XE programming environment is given. Then the program objects handled by the environment, modules, systems and generations, are described. The XE programming environment is tailored for the manipulation of these objects and it is impossible to use the environment without truly understanding their structure.

The fourth section describes the storage system, a kind of an object database, which hides the underlying file system, and is a basis for each tool in the programming environment. Then the different tools are described. For each tool both a tutorial introduction and a reference description is given.

2 Introduction

The purpose of the XE programming environment is to support most parts of the life cycle of XE programs. The XE programming language is described in *XE Reference Manual*. There are various facilities for the implementation, execution, testing, and debugging of programs. Only the earlier part of program development, i.e., the design and specification of programs, is not supported. The XE programming environment is highly integrated; it uses a dynamic linker and allows the target program to be changed while it is running, it incorporates an open and permanently active compiler whose data structures are accessible throughout the environment, and it keeps record of the components of the program and their relationships in order to automate many tasks that are normally performed by the user. It is based on an object database, which hides the underlying file system.

On the scale of *programming in the small - programming in the large* we have been more concerned of the latter. The programs that are developed are supposed to consist of many modules that may have complicated interdependencies so that the maintenance of the program integrity is a non-trivial task. There may also simultaneously exist many versions of the same program having different status and the programs may be developed by many programmers. Thus version management and the coordination of actions are supported to a high degree.

The level of expertise of the user is assumed to be that of a professional programmer. This implies that some of the more trivial tasks of program development, e.g., typing in a syntactically correct program, are not especially well supported. On the other hand, features which are critical in professional program development like restoring the consistency of a program after modifying a part of it are the main concern of the environment.

3 Basic Concepts

The top level syntactical constructions of the XE language are *constants, procedures, iterators, rules, and data types.* These are the basic entities manipulated in the programming environment and the generic name *module* is used to refer to them in this paper.

Every module has a name, a syntactic domain and a set of implementations. The name is an XE identifier. The syntactic domain is the name of one of the top level syntactic constructions of the XE language (see above). The elements of the set of implementations of a module are called *revisions*. The set can be partitioned into several *versions*. The versions differ from each other conceptually. They either perform a different task or the same task in a different way. The revisions of a version reflect the development of that version. They are the units of code that the programmer edits and that can be executed. Every revision has a (possibly different) *header*. The structure of the header depends on the syntactic domain of the module. For example, the header of a procedure consists of the argument types, the return types, the signals and the where clauses of the procedure.

A *system* is the topmost hierarchical component manipulated by the programmer in the XE programming environment. For example, a program or a set of programs that are used to perform a task can be a system. Another example is a graphic package that contains type definitions of various graphical entities and routines that manipulate them.

There can be many *generations* of the same system. The generations correspond to the historical development of the system and they form a tree. A generation associates with every module name that it uses a single implementation (some revision of that module). These associations are changed either manually or automatically as the system evolves. A generation has several attributes that describe its status of it, e.g., whether it is a released or an experimental generation.

Often it is necessary to use modules of another system as components in a system. To achieve this it is possible to *export* and *import* modules. A module is imported to the system if it is referred to in it, but not defined in it. Only modules, which are exported can be imported and only from systems which are *included* into the system, which refers to them. If a module could be imported from several systems *a name conflict* is said to arise and the user must resolve which binding he wants to use. The restrictions specified above as the import mechanism greatly reduce the amount of name conflicts.

As module names are associated to their implementations in generations of systems, the inclusion relation is extended to cover also this level ; i.e., a system generation may include specific generations of other systems. To further diminish the amount of potential name conflicts a system cannot include an other generation of itself and cannot include more that one generation of any other system.

4 The Storage System - the Object Database

The purpose of the storage system is to automate the storage and the manipulation of programs and their parts. The storage system is implemented on top of the file system and completely hides it. Both source code and binaries can be stored. However the internal data structures of the environment are not stored directly. Instead they are regenerated at the beginning of each programming session. This leads to highly economical use of storage at the expense of set-up time.

The objects stored in the storage system are top level modules, systems and generations. The top level modules are stored as source code files, as stripped source code files containing an entire generation, and as lisp-machine binaries in the *module pool*. The stripped source code files are files containing only the headers of modules. Systems and generations are stored in special description files, which display their development history.

4.1 The Module Pool

All XE code is stored in the module pool. Each module is represented by a file. The system is open in a sense that XE files from other systems can be inserted into it and extracted from it.

The module pool is organized according to the syntactic domain of each module and further according to versions and revisions. The pool is stored as a one to one map in the file system. The version, the revision and the owner information of a module are stored in the path name of the file containing the code. Normal file arguments like the creator, the write time etc. are also used. The fact that the pool is a one to one image of the file system guarantees good performance and excellent reliability but makes it difficult to add new features to it.

The module pool is used as a place to hold the contents of the user programs. However there are two other aspects of the module pool which make it independently interesting. It can be used both in search for a desired module and for storage reclamation.

Note that currently it is difficult to find the desired module from the module pool. The attributes given to modules lack structure and the user is hopelessly lost in middle of all XE code ever written. This problem is the usual information retrieval problem and it can be tackled only by creating a library system to organize the information. However the problem is not very serious. Usually code is reused by including other systems into the system under development and not by copying some arbitrary modules from the module pool. Then the systems, their documentation and the import and export mechanism creates the necessary structure to the unstructured module pool.

Even thought the module pool is not ideal for the retrieval of modules, it is quite suitable for reclaiming storage. The attributes make it easy, e.g., to delete all unreferenced revisions or to dump to a tape all revisions which are older than a specified date.

4.2 System and Generation Structure

When a new system is created, the system automaticly creates a generation for it. This root generation is an empty generation into which modules can be added. New generations of the system can be created by renaming an existing one and then changing it. In this sense the generation structure of each system forms a tree according its creation history. As each generation can include other generations they form another tree. The examination and manipulation of these trees is interesting mainly when seen from the viewpoint of the system manager. The systems and the generations are stored in the file system as direct history files, where the operations describe their development history. Updating operations are atomic so that integrity is maintained.

The programmer is more interested in the structure of the specific generation he is currently developing. The generation can be seen as a dependency graph, where the nodes are modules and the arcs correspond to "uses"-relations. The dependency relations are not stored. As the generation under development is compiled each time the programming session begins the information gets calculated anyhow.

5 The Tool Set of the XE Programming Environment

First a short introduction to the XE toolbox is given and then the different tools are described one by one.

The tool set of the XE Programming Environment consist of a ZMACS text editor, a compiler, an error message tool, an execution environment, and a module base browser. The editor, the error message tool, and the compiler are grouped together under a common interface 1 and called the program development tools.

Figure 1: The program development tools.

The module base browser differs from all other tools by its global nature, i.e., it tries to visualize all code in the programming environments database. The module base browser shows a flat module view. The modules are divided by their syntactic type and further by their development history. It allows easy access to the module data and makes it possible to copy modules between generations.

5.1 The program development tools

The common interface of the program development tools is divided into three panes: the editor pane, the error message pane, and the XE process pane. The editor pane is the input source of the ZMACS editor. All text and commands directed to the ZMACS editor are typed into it. The editor process is said to be associated with the editor pane.

As the XE system is based upon a module database, it needs to have some definite information of all modules inserted into the database. The information generated by the parser in sufficient and therefore only syntactically correct modules can be inserted into the database. This divides the errors found by the compiler into two different categories: the syntax errors and the semantic errors. The syntax errors must be corrected promptly because code which has syntactic errors can only be stored into the unstructured scratch area whereas the semantic errors can wait until it is convenient to pay attention to them. The system reflects this division by prompting for corrections of the syntax errors directly in the editor whereas the semantic error messages are written into the error message pane. The error message pane is automatically updated when the modules get compiled. The messages are mouse sensitive, clicking one selects the erroneous module into the editor buffer and points out the error location with the editor cursor.

The XE process pane is the primary output channel for the XE compiler. The compiler runs in a separate process and it doesn't communicate directly with the user. The pane exists only to inform the user of the status of compilations. The control commands to the compiler are usually given through the editor.

5.1.1 ZMACS editor

The primary interactive component of the XE programming environment is the ZMACS editor and its XE mode. Most operations concerning editing and compilation are performed under it. Only the commands which are specific to the XE mode are documented here. For a general

The buffers in the XE mode are divided according to the syntactic elements of the XE language. Each procedure, iterator, rule and data type has its own buffer. The constants in the generation under development are grouped generation wise. Of course the user can type several modules into one buffer, but when the buffer is saved it is parsed and divided into several buffers.

Code is accessed according to the module names. As names are unique within a generation no further specification is needed. Code can also be accessed in the *edit compiler warnings* mode or in the *edit callers* mode. These modes allow the user to step through specific points in the code.

The user can simultaneously develop several generations with many modules. To make the use of commands easier two defaults are used. The *current buffer* is the buffer that user is currently editing and the *current generation* is the generation that the compiler is compiling. Usually the commands are directed to these defaults. Most commands take a numeric argument, and when the argument is given they ask the target of the operation instead of using the default.

The most often used commands are boud to a specific key. The shorthands used for keys are:

sh - shift

c - control

m - meta

s - super

Keys can be combined. Eg. **s-m-sh-p** is typed by pressing down super, meta and shift keys and then pressing key p. The super, hyper, meta, control and shift keys are modifiers, they all act like a control key in a normal terminal.

Commands are:

This command is used to change the current buffer to edit the desired module. It can also be used to create a new module.

s-e *com-xe-edit-module*

> Loads the code of the given module into a buffer and selects it. Given an argument the command queries for the system and generation, otherwise the current generation is used.

These commands are used to store the code in the buffer to the database. The storage can be done either by incrementing the revision counter or by incrementing the version counter. The version counter should be changed only when the module in the buffer greatly differs from the previous one. The buffer must be parsable for the operation to succeed.

s-r *com-xe-new-revision*

Stores the modules in the current buffer to the database incrementing their revision counter. If there are several modules in the buffer, new buffers are created for them.

s-v *com-xe-new-version*

Stores the modules in the current buffer to the database incrementing their version counter. If there are several modules in the buffer, new buffers are created for them.

This command is used to change the status of a module so that it can be accessed outside the generation it is defined in.

s-x *com-xe-export-module*

Exports the module in the current buffer. Given a numeric argument ask for the generation and module name, and then exports them.

These are the compilation control commands. They allow to check a single module, to check a whole generation and to generate code.

s-c *com-xe-check-module*

Queries a module name of the current generation and checks the module. Given a numeric argument asks the generation.

s-sh-c *com-xe-check-modified-definitions*

Checks all modified definitions of the generation of the current buffer. If current buffer doesn't have a generation associated to it checks the modified definitions of the current generation.

s-g *com-xe-codegenerate*

Generates code for the current generation.

A information retrieval command: **s-i** *com-xe-info*

Prints the name of the current generation.

The following commands control the edit error messages mode. **s-c-e** *com-xe-edit-error-msgs*

> Initiates the edit error messages mode by dividing the window and showing the error messages in the upper one and the code in the lower one, and positions the cursor to the location of the first error.

s-c-n *com-xe-edit-next-item*

> Positions the cursor to the location of the next error message.

s-c-n *com-xe-edit-prev-item*

> Positions the cursor to the location of the previous error message.

s-c-q *com-xe-quit-editing-items*

> Ends the edit error messages mode.

The next commands are not bound to any key. They can be given as extended command by typing *m-x command-name*. The first set concerns the creation of systems and generations, and the relations of generations. **m-x xe create system** *com-xe-create-system*

> Creates a new system and starts to edit it first generation.

m-x xe create generation *com-xe-create-generation*

> Creates a new generation of a system and starts to edit it. Asks the father generation.

m-x xe edit generation *com-xe-edit-generation*

> Asks for a generation, loads it if necessary and starts to edit it.

m-x xe include generation *com-xe-include-generation*

> Asks for a generation, loads it if necessary, and includes it to the current generation. Given a numeric argument asks first a generation and includes to that generation instead of the current generation.

The commands, which handle xe modules as a tag table. **m-x xe select generation as tag table** *com-xe-select-generation-as-tag-table*

Selects an XE generation as a tag table.

m-x xe select loaded generations as tag table *com-xe-select-loaded-generations-as-tag-table*

Selects loaded XE generations as a tag table.

The last commands are seldom used. They allow user to export all modules of a generation and to clear the state of the compiler. **m-x xe export all** *com-xe-export-all*

Exports all modules of the current generation. Given a numeric argument asks a generation and uses that instead of the current generation.

m-x xe delete buffers *com-xe-delete-buffers*

Deletes all buffers, which are in XE mode.

m-x xe clear-all *com-xe-clear-all*

Clears all data structures of the compiler.

5.1.2 The XE Compiler

The XE compiler runs in a process of its own. Its operation is controlled by the commands given to the ZMACS editor and by the changes made to the source code. When an check modified definitions generation command is given the compiler searches for new or changed definitions and compiles them and the definitions affected by the changes.

5.1.3 The Error Handling Facility

Because the database can only contain syntactically correct code, syntax errors require immediate notice. The most convenient way to correct them is to use the edit compiler warnings command in the ZMACS editor.

The other errors found by the compiler, the semantic errors, are handled differently. They appear in a code that has been successfully parsed and therefore they can be associated with a revision of a module. The error messages concerning them are inserted into the error message pane, and the erroneous positions can be edited at will by clicking the mouse sensitive error messages. As modules get recompiled the old error messages associated with them are removed from the error message pane and the new ones, if any, inserted into it.

5.2 The Execution Environment

Currently the execution environment doesn't differ from the lisp listener. An interface of its own should be considered first when its functionality is enlarged.

Function **exec** &optional *(string (cg$get-xe-line))* &key *(showp nil))*

> Exec takes an XE statement and executes it in the Lisp Machine. The statement is compiled into a routine and this routine is called after its lisp code generation has been performed.

Function **execs** &optional *(string (cg$get-xe-line)))*

> Execs takes an XE expression and compiles it into a routine which will produce the value of the expression to the standard output stream.

Function **xe-trace** *name*

> Sets tracing on in the routine named *name* (a string), i.e., each entry and exit of the routine will be reported.

Function **xe-untrace** *name*

> Sets tracing off in the routine named *name* (a string).

5.3 The Module Base Browser

The module base browser shows a flat view to all XE code ever written under the control of the XE system. The module base is partitioned at the topmost syntactic level. The resulting fragments are parsable top level XE definitions.

Module fragments are divided by their syntactic type and their development history, i.e., each module of a given syntactic type is divided into versions and revisions. The fragments have a few attributes associated with them. The most important are the owner generation, i.e., the generation which uses this fragment. Each fragment can have at most one owner. Most fragments have none.

The module base browser has two different uses: reuse by copying and storage reclamation. Reuse by copying means searching for a desired piece of code, to be copied into the generation under development. Currently it is not easy to find the desired code, because there is no library mechanism to bring order into the chaos of all code ever written. The storage reclamation aspect suffers a bit from the same problem. However the problem is not as severe, because it is sufficient to reclaim most of the referred storage and the ability to find the code not owned by any generation usually does the trick.

5.3.1 Usage

The module base browser is initiated by typing `<select>-w`. Its usage is quite straightforward. The icons in the left column represent the syntactical categories of the XE language 2. Clicking one of them shows the names of the code fragments of that category. The names are also mouse sensitive. A left-click on one of them will pop up a temporary window which shows how the fragments are divided into versions and revisions.

On this level a single code fragment can be identified and various operations performed on it. A code fragment can be viewed, edited and copied by clicking left on it and then choosing from the menu. The copy operation queries for the system and the generation to copy into.

Storage reclamation can be performed any time by giving the storage reclamation command. It allows one to delete all code which is not part of any generation. The deletion can be further restricted using the attributes: user, creation time, syntactic category etc. A code segment which is part of a generation can only be deleted by deleting that generation.

Figure 2: The Module Base Browser

Contents