(Marti83) Robert W. Marti. Integrating Datbase and Program Descriptions Using an ER - Data Dictionary from Entity - Relationship Approach to Software Engineering, C.G. Davis Ed., Elsevier Science Publishers, 1983, pp. 377-391.

(Miller88) Glenn Miller, Mark Johnston, Shon Vick, Jeff Sponsler, and Kelly Lindenmayer. Knowledge Based Tools For Hubble Space Telescope Planning and Scheduling. Telematics and Informatics, Vol. 5 No. 3, 1988, pp. 197 - 212, Pergammon Press.

(Narayan88) Rom Narayan. Data Dictionary Implementation, Use and Maintenance. Prentice Hall, 1988, pp. 77 - 81.

(Partridge86) D. Partridge. Engineering Artificial Intelligence Software. Artificial Intelligence Review, Vol. 1, No. 1, 1986, pp. 27-41.

(Sheil83) Beau Sheil. Power Tools for Programmers from Readings in Artificial Intelligence and Software Engineering, Charles Rich and Richard C. Waters Eds., Morgan Kaufman 1986

(Steele84) Guy L. Steele. Common Lisp The Language. Digital Press, 1984

(Bershad88) Brian N, Bershad. A Remote Computation Facility for a Heterogeneous Environment. IEEE Computer, May 1988, pp 50-60.

(Bucher75) Bucher. Maintenance of the Computer Sciences Teleprocessing system, from Proceedings of the International Confernce on Reliable Software, 1975.

(Cichinski88) Steve Cichinski and Glenn S. Fowler. Product Administration Through SABLE and NMAKE. AT&T Technical Journal, July/August 1988, pp 59-70.

(Dhar88) Vasant Dhar and Matthias Jarke. Dependency Directed Reasoning and Learning in Systems Maintenance support. IEEE Transactions on Software Engineering, Vol. 14, No. 2 , Feb. 1988, pp 211-227.

(DeMichiel89) Linda G. DeMichiel. Overview: The Common Lisp Object System. Lisp and Symbolic Computation, 1, pp 227-244, Kluwer Academic.

(Erman88) L.D. Erman, J.S. Lark, and F. Hayes-Roth. ABE: An Environment for Engineering Intelligent Systems, IEEE Transactions on Software Engineering, Vol. 14, No. 12 , Dec. 1988, pp 1758-1770.

(Fairley85) Richard E. Fairley, Software Engineering Concepts, McGraw Hill, 1985, pp. 284 - 285.

(Feldman83) S.I. Feldman. Make - A Program for Maintaining Computer Programs from the Unix Programmers Manual, 7th Ed. , Bell Tech. Labs. , pp. 291 - 300.

(Freedman85) Roy S. Freedman. Programming with APSE Software Tools. Petrocelli Books, Princeton, New Jersey, 1985.

(Godfrey85) M.D. Godfrey, D.F. Hendry, H.J Hermans and R.K. Hessenberg. Machine- Independent Organic Software Tools (MINT). Academic Press, 1985.

(Hatton88) Les Hatton, Andy Wright, Sturat Smith, Greg Parkes, Paddy Bennett and Robert Laws. The Seismic Kernel System - A Large Scale Exercise in Fortran 77 Portability. Software Practice and Experience, Vol. 18 (4), (April 1988)pp 301-329 .

(Keene89) Sonya E. Keene. Object Oriented Programming in Common Lisp, Addison Wesley, 1988.

(Kruse87) Robert L. Kruse, Data Structures and Program Design, Prentice Hall, 1985, pp 407 - 411.

```
(SHOW-MARIAN-HISTORY (&key verbose (sort-by :time))
```

10. Future Extensions to the Tool Set

Planned extensions to this tool set include the integration of an intelligent data dictionary. with the librarian facility. (Narayan88) The data dictionary would be based on the notion of software objects that are tied together with an Entity Relation Model (Marti83) and implemented as a semantic net similar in construction to REMAP (Dhar88), So for example when a user uses Marian to checkin a source file (a software object) the file would be scanned for function, special variable and defclass definitions. These objects would be linked in the model to the source file by a *:is-contained-in* link. The system could follow the the *:is-tested-by* links constructed by the LTM to decide what tests should be run to assure the system correctness.

As a further example the :is-called-by and :is-contained-links links could be analyzed to diagnose possible problems with the system definitions important to the system builder. This last example is similar to the tool set integration between SABLE and NMAKE. (Cichinski88).

Further source files are checked in the comments and documentation strings for software objects like functions,methods, etc could be put through a hypertext type of authoring mechanism. This would allow intelligent perusal of software and perhaps promote software reusability.

11. Conclusion

This paper has discussed a set of tools and techniques used in the development of Common Lisp software in a heterogeneous machine environment. The test manager, system builder, code transfer and archival mechanism as well as a librarian facility have been ported to a number of machines and implementations. These tools are implemented in only Common Lisp and the Common Lisp Object System (CLOS). The methodology of utilizing a machine independent file path construct was also described. The tools described here have ported with relative ease (less than one days work) to a number of different machines. Porting becomes a matter of assuring that a very small core of functions that define operations on the machine independent path abstract data type work. We are now directing our efforts towards expanding the tool set with the creation of intelligent dependency analysis operations as described in the further extensions section.

12. References

Here is the definition of several of the cataloged functions above with a brief explanation of their functioning:

```
(CHECKOUT (name &key comment
              (from (marian-lib))
              (type *lisp-file-type*)
              (to (pwd))
              (verbose (marian-verbose)) )
```

The form of the from argument from may be either the same as for a machine-independent path directory (that is a list of atoms, where an atom is either a string or a symbol), a reference to a machine independent logical (see show-translations) or a machine dependent-path (i.e a namestring). If the form of the argument is a machine dependent path then only the directory part of the path is used. The form (setf (marian-lib) ...) is used to set the default value of from, e.g. if you are checking out several files from one library. The to argument specifies where the file should be moved. It defaults to the present working directory (see pwd and cd). The to argument may be given as a machine dependent or machine independent path or a list of directories. If verbose is true the function displays what actions it is taking. If checkout was successful, the function returns the path where it put the file otherwise it returns nil. Here is an example:

```
       (cd "vick.work;")
       (setf (marian-lib) $utilities)
       (checkout 'general-utilities
           :comment "shows use of cd and marian-lib for to
```

```
(CHECKIN (name &key comment
 (from (pwd)) (type *lisp-file-type*) to (verbose (marian-verbose)))
```

Checkin looks at the reservation files to assure that the source has been reserved. It deletes the reservation and detaches the semaphore. If a user wants to cancel a reservation or break the reservation lock for a source reserved by another user he or she may do this with the unreserve function.

```
(UNRESERVE (name  &key comment (type *lisp-file-type*)
                 (verbose *marian-verbose*) force)
```

Unless the force flag is non-nil a reservation must exist for the file being unreserved.

The functions that show the reservations or history both take arguments to tell how the history should be sorted and what history records should be returned.

```
(SHOW-RESERVATIONS (&key (name :all) (type :all)  (reserver :all
                         (verbose (marian-verbose))
                          (sort-by :time))
```

Here are a few examples

```
> (ship-files "spike:spike.develop;*.*" :only 'lisp)
```

will put all the lisp files in develop directory structure into the holding area on the Scivax machine. Whereas,

```
> (ship-files "spike:spike.develop.chi;*.*"
      :only 'lisp :recursive nil)
```

will put only the chi sub-directory code into that holding area.

9. A Librarian Facility

We have developed a librarian tool called Marian. The main purpose of Marian is to prevent this problem of uncoordinated source file changes through a file reservation system. Before a developer changes a file, she or he must first execute a command which reserves the file in his or her name. Should another developer attempt to reserve the same file, Marian will detect the collision and inform the second developer. Another benefit of Marian is that it keeps track of what files have been changed so that we can ensure the quality and document each software release .

Marian uses a reservation file as a semaphore. When a user of Marian wants to checkout a file Marian obtains the reservation file semaphore in the name of the user, adds a reservation entry and writes the file back to disk. A history record is also appended to a history.

The interaction with Marian can be specified relative to current marian-lib which can be specified in either a machine dependent or independent way, with or without logical translations. Marian stores all path information internally in a machine independent way so that the reservation and history files may be shared across machines.

The following is a brief catalog Marian functions:

• checkout (or check-out)	reserve a file in your name
• checkin (or check-in)	put a new or modified file in the library
• unreserve	remove a reservation
• show-reservations	what's reserved
• show-my-reservations	what's reserved by you
• where-tis	locate a file in the library
• show-marian-history	history
• marian-lib	to see or set the current default library
• marian-verbose	to see or set the default verbosity level

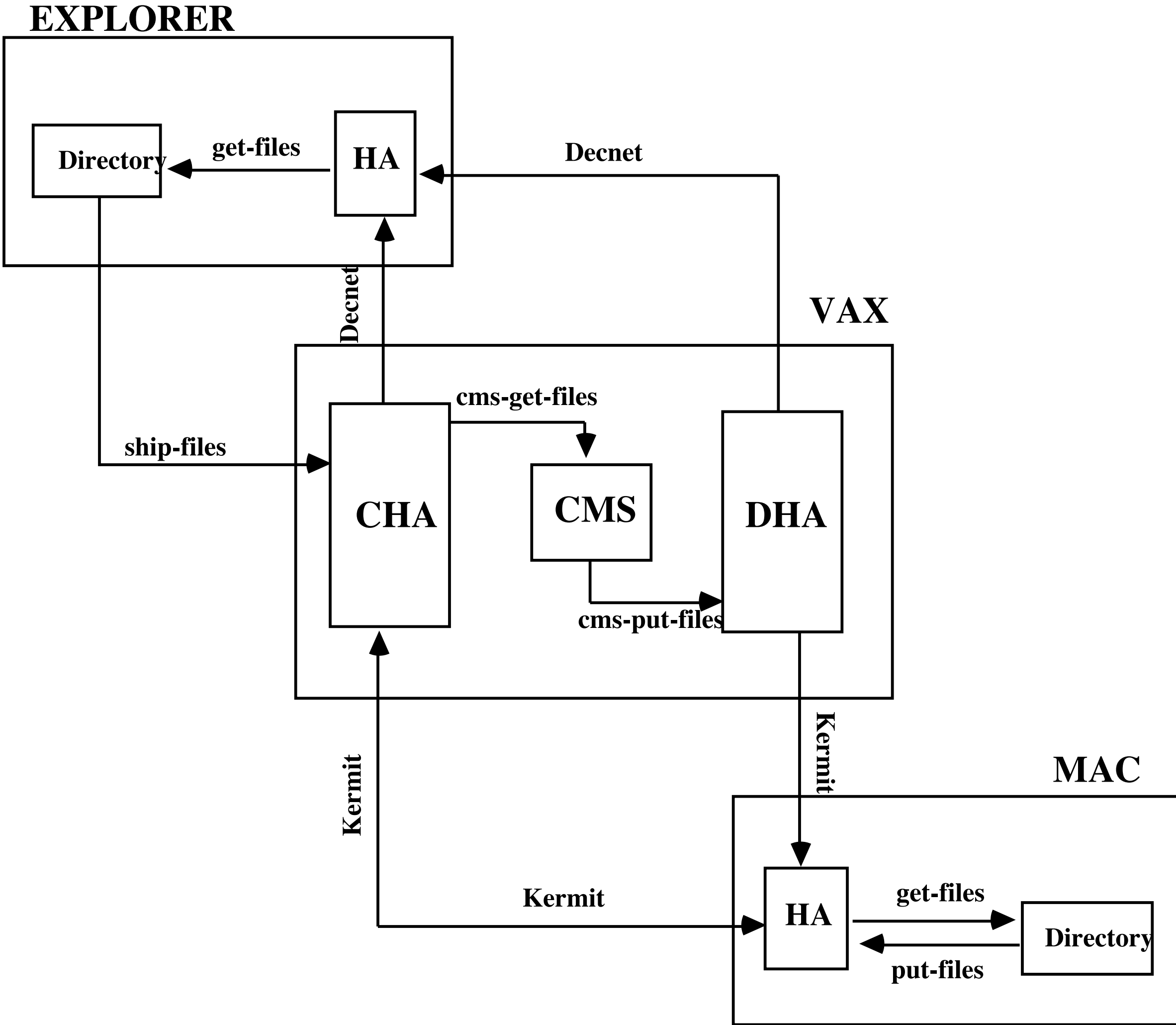
8.1.1. Using the ship-files function.

The flavor of the functions cataloged above will be illustrated by a more complete definition of the ship-files function. This tool takes a file or set of files and ships them to the holding area on another machine. It can only be used to transfer files between machines which support TCP/IP or Decnet.

SYNTAX:

```
(ship-files path &key (to 'scivax)(recursive t)
      exclude only since (confirm t) cms
      (version *CURRENT-VERSION*) )
```

- path — identifies the file(s) to be transferred. Wildcarding is allowed. Path is in the syntax of the machine of the local machine
- to — identifies where the files are to be shipped
- recursive — whenever a directory exists as part of the pathname, do a ship-files on it as well. The default is for ship-files to recurse.
- exclude — either an atom or list and specifies the type of files to exclude in the shipping process, e.g. `:exclude 'dat` would not ship any .dat files.
- only — argument can be either an atom or list and explicitly states the file type to be transferred, e.g. `:only 'lisp` would ship only .lisp files.
- since — only files with a creation date greater to or equal to the value of this keyword will be processed.
- confirm — if this keyword is t (and this is the default value) the user is prompted for each file. The confirmation happens via a popup menu on machines that support it unless the value of the keyword is :tty in which case a prompting sequence ensues.
- cms — if this keyword is nil, then cms-get-files will ignore these files when adding files to the CMS libraries. This is useful for development and testing purposes. Note that the default value is nil, so that you must explicitly set this flag if the files are to eventually be entered into CMS.
- version — this specifies the corresponding version. The value should be a version data structure that is not discussed here. This keyword is used by CMS library loading functions.



The code transfer and archival system detailed here is also integrated with the Code Management System (CMS) package in VMS on the Vax. By setting a keyword flag a user can specify that the files should be loaded into the CMS system. There are functions that take the contents of the holding area and load it into the CMS directory structure (relative to a CMS root directory) as well as functions that do the inverse, going from the CMS libraries to the holding area.

- delete-files-from-holding-area - removes all files from the holding area
- describe-note-filesprints contents of file.notes
- clean-up-note-files- removes any references in file.notes to a file which is not in the holding area (usually due to network problems)
- clean-up-msg-files - deletes any message file which is not referenced by a notes file
- clean-up-holding-area - performs clean-up-msg-files and a clean-up-note-files
- load-netestablishes network information

The functions ship-files and put-files populate holding-areas. They each take a path argument that identifies what part of the directory structure to collapse into the holding area. They create what is called a MSG file for each file along the given path and a NOTES file that describes the mapping between the encoded MSG file name and the actual name and location of the file. These NOTE files then contain pairs of encoded MSG file name and MIPS that describe the actual name and location of the file.The function get-files which retrieves the files from the holding-area and moves them into the regular directory structure just read the NOTE files and does the reverse mapping.Note that there is a concept of a root directory. Thus the directory structures on the different machines need not be absolutely the same on the same relative to a root directory. Suppose that the root directory on the VAX is disk$bruno:[ai_library.spike] and there is a MSG file that corresponds to an Explorer file named:

```
spike:spike.develop.chi;commands.lisp#29
```

Now the representation of this file that appears as the NOTES file is:

```
#S(machine-independent-path :host "spike" :device " " :directory
("spike" "develop" "chi") :name commands :type "lisp" :version 29)
```

The functions that operate on the holding area on the VAX understand that this becomes the file disk$bruno:[ai_library.spike.develop.chi]commands.lisp

The figure illustrates how the various commands transfer files within a machine and between machines in the particular we had at one point during the Spike project, The arrows are labelled with the functions used to transfer files. The system illustrated contains just 3 computers: an Explorer, a VAX and a Macintosh. The terminology employed here is similar to that used in THERE system. (Bershad88)

```
compile-date(B) := check-recursively(B)
; post build actions for B if necessary
if compile-date(B) > compile-date(A) and
   compile-request(A) = T
   then post (:compile A) action at time=NOW
else time=compile-date(A)
if load-request(A) then post (:load A) action
return time  ; used for other comparisons
```

7.3.8. Executing Build Actions

There are two unusual situations in this part of the algorithm. If the user has specified selective then loop over the list of build actions and ask for confirmation about each. If the user has specified *print-only* then print the list of build actions and quit. The usual case however is that each build action in the list in slot build-actions is interpreted. If the build action is :compile the source file for the module is compiled. If the build action is :load the object file for the module is loaded.

8. Code Transfer and Archival

In a multiple machine environment it is necessary from time to time to synchronize and distribute the release of a software system. This section describes a tool that allows a software system to be archived and then distributed to different machines. The basic idea here is that at some point a directory hierarchy is put into a flattened form. This flattened form is then transmitted via some communication protocol (TCP/IP, Chaosnet, Kermit, or even tape) and the directory hierarchy is resurrected.

8.1. Holding Areas

Each machine has a “holding area” (HA) where files are temporarily staged. On all machines, the holding area is used to receive files from another machine. The files are then moved from the holding area to the correct directory tree. The holding area contains the flattened directory hierarchy with special files that instruct the system how to resurrect the hierarchy.

Here is a brief catalog of the functions available to the user:

- ship -files - copies files from local machine to holding area on another machine (for machines which support Decnet or TCP/IP)
- put-files - copies files to the holding area of the local machine (for machines which do not support Decnet or TCP/IP)
- get-files - copies files from the holding area into the proper directory (inverse of put-files)

Consider this part of a system definition:

```
(:compile-load spike-core (:depends-on globals))
```

A compile action specification such as the one above is interpreted as follows: The **:compile-load** specification causes a T to be placed into the *compile-request* and the *load-request* slots of the module *spike-core*. If the specification had been **:compile** then the load request would not have been noted. The symbol **globals** is used to key into the module-catalog and the resulting module object is placed in a list in the slot *load-first*.

7.3.6. The Module Network

As a result of interpreting the system definition zero or more module hierarchies will result. Each hierarchy will have a root node which depends on no other module (but may be depended on), leaves that have dependencies, and intermediate nodes that have both characteristics. Each tree should be free of cycles and so an exhaustive examination of all nodes is done to verify this. The algorithm is simple: Each module is touched only once (a list of nodes is kept to make sure of this) and labeled with a unique marker which is used to see if the algorithm has passed through the module and returned to it (revealing a cycle). If a module has already been marked, an error is flagged indicating the presence of the illegal cycle. If the module has not been marked and it is not in the list of previously checked modules, then each of the modules in the load-first slot is checked.
The next step that is done is to loop over all modules in the system and for each create the true pathnames from the *system-logical-host*, the module logical *directory*, and the module's *file*.

7.3.7. Collecting Build Actions

In order to build a system, Build System must collect the set of build actions. It does this work on the basis of the file information associated with each module. Each time that a build-system is done, every module must be updated as to the status of its own associated files. The program loops over all modules, and for each the file system is checked for the appropriate file information. If numeric version is the criterion then the algorithm is this:

```
; Assume depends-on(A,B)

if not-loaded(B) then post (:load B) action
if version(objectfile,A) < version(sourcefile,A) and
   compile-request(A) = T then post (:compile A) action
if load-request(A) then post (:load A) action
```

If the criterion for file recency is creation date, the algorithm is this:

- The System Definition File must be located and loaded to create the system objects and network of module objects.
- The module network is traversed in order to determined the set of necessary Build Actions.
- The Build Actions are interpreted.

7.3.4. Finding the System Definition File

When the `(build-system 'system-symbol)` function is evaluated, first a check is made to see if the system has been created before. This is done by looking at the variable `*TABLE-OF-SYSTEMS*` which should be bound to a hash table. Using the *system-symbol* as hash key, the desired value is an object of the class *system-table-entry*.

1. If there is an entry for *system-symbol* return it. Otherwise make an entry and store it in the table.

2. See if the most recent version of the System File has been loaded (load it if necessary). This check is based on numeric version and is important if the System Definition File is moved.

3. Determine whether the most recent version of the System Definition File has been loaded. If the correct System Definition File has not been loaded, it is loaded and the system definition stored there is evaluated. If the keyword option **definition-reload** is NIL, the System Definition File will not be reloaded even if a newer version is found.

4. The pointer to the new system object is stored in the table entry object and returned.

7.3.5. Creating the System and Modules

As mentioned in a previous section, when the System Definition File is loaded and the system definition form evaluated, a set of objects is instantiated. This set includes the unique system object and its modules. From the definition, the name, package, and logical-host are determined and stored in the system object. The system object slot *module-catalog* will contain a hash-table that provides a mapping from module names to their objects.

Each module specification that is encountered causes a *module object* to be created and stored in the *modules* slot of the system object. The symbolic name of the module is extracted and stored in the object. The machine-independent file information is at this point dissected into the directory and file portions and those are stored in the corresponding slots in the module object.

7.3.1. Class: System-Table-Entry

```
(defclass SYSTEM-TABLE-ENTRY ()
  (system-file               ; Points to definition file
   system-file-version       ; Check for recency
   definition-file           ; System definition storage
   definition-file-version   ; Recency of definition
   object))                  ; System object
```

7.3.2. Class: System

```
(defclass SYSTEM ()
  (name                                 ; Symbol id for system
   modules                              ; List of modules
   module-catalog                       ; Table of modules
   (package :initform 'user)            ; Default package
   (logical-host :initform "SYS")       ; Logical file host
   (file-version-check :initform :by-version)
   (load          :initform t)          ; Load if necessary?
   (compile       :initform t)          ; Compile?
   (recompile     :initform nil)        ; Compile in any case?
   (print-only    :initform nil)        ; Inform only?
   (silent        :initform nil         ; Suppress output
   (selective     :initform nil)        ; Ask the user?
   (build-actions :initform nil))       ; List of actions
```

7.3.3. Class: Module

```
(defclass MODULE ()
  (name                      ; Symbolic id of module
   system                    ; Parent System instance
   (directory :initform "")  ; Logical directory string
   (filename :initform "")   ; File string
   pathname                  ; Translated source file
   compiled-pathname         ; Translated object file
   true-pathname             ; Source file pathname
   true-compiled-pathname    ; Object file pathname
   (date :initform 0)        ; Universal time compilation
   compile-request           ; Compile module's file
   compiled-flag             ; Most recent file compiled
   load-request              ; Load module's file
   loaded-flag               ; Object file has been loaded
   recompile-request         ; Compile the file in any case
   load-first                ; Dependency modules
   cycle-flag))
```

These are the main steps that are taken during a system build.

The *keyword options* are described below.

• **compile** - If non nil modules are examined and for a given module, if its compiled file version is out-of date, then a compile module build action is noted for that module.
• **recompile** - If non nil the source file associated with each module in the system will be compiled regardless of version.
• **selective** - If non nil the system builder will query the user for compile and load confirmations.
• **print-only** - If non nil the system modules will examined, build actions collected and printed to the screen. No compiles or loads will be effected.
• **file-version-check** - May be either **:by-version** or **:by-date**. These two options produce very different behavior. The **:by-version** value is the default and is responsible for causing the numeric file version numbers to be considered when testing a module for having any out-of-date object file; the testing will be intra-modular and will not usually affect other modules. The **:by-date** value will cause the file creation dates to be used for this test.
• **silent** - If non nil then the messages that indicate build-action executions are not printed.

Here are a few small examples of invoking the system builder on a TI Explorer:

```
(build-system 'spike-develop :compile t :print-only t )

Build Actions:
-------------
Load module: GLOBALS
    file: Spike: SPIKE.DEVELOP;SPIKEGLOBALS.XLD#6
Load module: SPIKE-CORE
    file: Spike: SPIKE.DEVELOP.CORE;SPIKE-CORE.XLD#12
Compile module: PLANNING-SESSION
    file: Spike: SPIKE.DEVELOP.CORE;PLANNING-SESSION.LISP#16
Load module: PLANNING-SESSION
    file: Spike: SPIKE.DEVELOP.CORE;PLANNING-SESSION.XLD#16
```

This example below illustrates the **selective** option; the user is asked to affirm each build action. No logical checks are done here; that is, if a user denies a request to load a module upon which other modules depend, errors may occur.

```
(build-system 'spike-develop :selective t)
Load module: GLOBALS? (Y or N) Yes.
Load module: SPIKE-CORE? (Y or N) Yes.
Load module: PLANNING-SESSION? (Y or N) Yes.
```

7.3. The Data Structures and Algorithms Behind the System Builder

The main classes used are *system-table-entry* (used to keep a record of the system description files), *system* (used to describe the organization of the files for a single application) , and *module* (used to describe one component of a system).

A small example of s system definition follows:

```
(define-system SPIKE-DEVELOP
   (:name "Spike Develop")
   (:package spike )
   (:logical-host 'spike)

   (:module globals           ($develop spike-globals))
   (:module spike-core        ($core spike-core))
   (:module planning-session  ($core planning-session))

   (:compile-load globals)
   (:compile-load spike-core        (:depends-on globals))
   (:compile-load planning-session (:depends-on spike-core))
```

The **define-system** macro can be divided logically into three parts: *major system attributes, module declarations,* and *module build-actions and dependencies*. The ordering of these parts is important; modules declared in the second section are referenced in the third section. The ordering of the declarations within each logical section is not important.
The *major system attributes* include the name of the system, its package (the symbol space where all compiling and loading will occur), and the logical host specification (the prefix that identifies logical pathnames).
The *module declarations* section contains forms that have the syntax: `(:module` *`module-name`* `(`*`logical-directory file-name`*`))`. Once a module has been defined, it can be used by name in the *module build-actions* section.

The *build-actions* section may contain forms with the syntax:
(*build-action module* [(:depends-on {*module*}*)])

The build-action must be one of :**compile** or **:compile-load**. The **:depends-on** specification is optional and will be interpreted to mean that before the build action is effected, the modules named in the specification must first be compiled and loaded into Lisp memory. In the example, assume that the module *planning-session* depends on *spike-core* and *globals*. The *planning-session/globals* dependency will be inferred (*planning-session* depends on *spike-core* which depends on *globals*) and therefore does not need to be explicitly stated. Transitivity of module dependency is supported in this implementation.

7.2. The System Builder's point of view

The System Builder may cause a system to be built by calling the function: **build-system**. The syntax for the call is:

```
(build-system 'system-name  &key
     recompile selective print-only silent (compile t)
     (file-version-check :by-version)(definition-reload t)
```

The System Builder provides the tools and programs to serve two different users: the System Designer, who has knowledge of the correct file locations and dependencies, and the System Builder, who wishes to call a single program that will create a system that he can use. The System Builder may be a program developer or an end user. Utilities have been defined that provide the System Designer with the ability to define the modules in a system and the dependencies of those modules The System Builder is provided with the utilities to cause a defined system to be built; this build includes the compilation of files that do not have up-to-date object versions and the correctly ordered loading of object files into Lisp memory.

7.1. The System Designer's point of view

This section discusses Build-System from the System Designer point of view. The System Designer is responsible for three main tasks.

- Defining the translations for the system.
- Defining the system.
- Storing the definitions in a file in the correct place.

There are two important files that must be created and placed appropriately: the **System File** and the **System Definition File**. The latter file stores the definition of the system (modules and relationships); the former stores a mapping from the system symbol to the System Definition File. The System Designer places the System File in the top level directory **"build-system".** There should be one such file per system and it should contain a form that has this syntax:

```
(system-definition-file 'system-name system-definition-file)
```

The system definition file can be a machine independent file reference as illustrated below:

```
(system-definition-file 'spike
 (create-machine-dependent-path
      :host 'example
      :directory '(spike system-defs)
      :name 'spike
      :type 'system)
```

This code defines the location of the system definition to the the System-Builder and allows one to merely request a build for the *system name*. In the System Definition File the system designer places the lisp forms that define the logical translations desired and define the system and its modules. Other forms may also be present to create the system package, set up special variables, and so on.

```
  (*OUT-2* "Spike:SPIKE.DEVELOP.TESTING.RESULTS;OUT.RESULTS#>"
   :DIRECTION :OUTPUT
   :IF-DOES-NOT-EXIST :CREATE
   :IF-EXISTS :SUPERSEDE)
      (LET* ((*OUT-1* (MAKE-SYNONYM-STREAM '*OUT-2*)))
        (WITH-OPEN-FILE
         (*OUT-3* "Spike:SPIKE.DEVELOP.TESTING.RESULTS;OUT3.RESULTS#>"
                                :DIRECTION :OUTPUT
                                :IF-DOES-NOT-EXIST :CREATE
                                :IF-EXISTS :SUPERSEDE)
          (LOAD
              "SPIKE: SPIKE.DEVELOP.TESTING.TESTS; COMPLICATED.TEST#>"
                :VERBOSE NIL
                :IF-DOES-NOT-EXIST NIL)))))
(LOAD "SPIKE: SPIKE.DEVELOP.TESTING.TESTS; EPI.EPI#>"
    :VERBOSE NIL :IF-DOES-NOT-EXIST NIL))
```

6. A Machine Independent System Builder

This section describes the Build-System tool which has been created in order to support the organization and building of large Lisp programs. It has been implemented using Common Lisp and CLOS. The Build-System tool is an automated facility for managing the files in a large system in a machine-independent manner. It includes a protocol for creating a system definition. This definition encodes the system designer's knowledge of the files in a system and is used to guide the compiling and loading of the system files. Files are represented as modules. Information in the definition specifies what modules are to be compiled or loaded and for a given module what modules must be compiled or loaded first in order to minimize the possibility of referencing undefined macros, functions, special variables, and data structures. A flexible external function provides the ability to build systems and has several useful options. This facility is similar in flavor to a number of software tools. These include the Unix make command (Feldman83) and the make-system command available on many Lisp machines. Our implementation differs in several ways. First in some of the other systems transitive module dependency analysis is not supported. That is, in some of the other systems if C depends on B and B on A, the dependency information for C must explicitly indicate that a pre-load of A and B (in that order) is required in the specification. Our builder performs the dependendency closure without explicit specification in the system definition. Further all the other cited systems are machine and implementation dependent. Ours is entirely machine independent. The system definition files reference module location using machine independent logicals and paths. Finally our system supports the concept of subsystems.

7. The System Builder from two different points of view

• define-test-suite - associates a set of one or more tests with a name
• undefine-test-suite - undoes define-test-suite
• describe-test-suite - displays brief description of tests in suite and suite info

Creating and Looking at Test Results
• run-test - runs tests and captures results of evaluating test forms
• run-test-suite - runs the set of test associated with a suite
• compare-results-to-benchmark -analyzes difference of test form results to benchmark
• update-benchmark - makes the current results file the benchmark and resets results
• examine-test - displays information on the status of tests, and whether referenced files exist
• move-tests - moves tests, prologs, and epilogs into the testing structure from a user directory

Operations On the Test Table
• load-test-table - loads the test table into memory
• save-test-table - saves the test table to disk
• clear-test-table - clears the in memory test table

5. Design Notes for LTM

At the heart of the system is the test table data structure. This is implemented as a hash table whose key is a canonical representation for a test or test suite. The value stored for the key is a structure that contains the test name and description, the mappings and MIPs that point to the location of the files that contain the test, epilog and prolog forms. The mappings are also stored internally as MIPs although when they are defined by a call to define-test they may be declared in either a machine dependent or machine independent fashion.

When the test-table is stored on disk the values of the table are used to construct calls to the definition functions. That is the test table on disk does not contain raw data structures but rather the the calls to the functions to resurrect the data structures. In this way the underlying data structures used in LTM can change without invalidating the test table files.

When the macro *run-test* is invoked the code to run a test is created. The basic idea is to create an *unwind-protect* that first loads the prolog,then loads the test file, and then as the protected form, loads the epilog file. The following example using the definition of the test named *complicated* from the define-test section above shows the code created from the *run-test* macro.

```
(UNWIND-PROTECT
 (PROGN
  (LOAD  "SPIKE: SPIKE.DEVELOP.TESTING.TESTS; PRO.PRO#>"
      :VERBOSE NIL :IF-DOES-NOT-EXIST NIL)
   (WITH-OPEN-FILE
```

4. Defining, Running and Comparing Results of Tests

The general steps for testing in LTM are to prepare a test, define it to LTM, run the test, compare the results of the tests to a benchmark and possibly update the benchmarks. A test is established within LTM with the define-test macro. The purpose of this macro is to create a definition of a test and enter it in the test-table.

Syntax:

```
(define-test test-name
  &key test-file mapping prolog epilog description)
```

The test-file argument is the name of the file that contains the test forms. The mapping is described above. This name component is either a string or a symbol and refers to a file that will be created in the results directory. Note that if no mapping is given explicitly with a keyword argument there is an implicit mapping for *standard-output* to a *result* file.

```
Example :

(define-test complicated  :test-file comp
:description "this is a fairly complicated example"
:mapping ((*out-1* out)  (*out-2* out)  (*out-3* out3))
:prolog  pro  :epilog   epi )
```

will set up a test named `complicated` in the test table. It will associate the file comp.test, pro.prolog, and epi.epilog in the tests directory with the test as the test, prolog, and epilog files respectively. It will direct all output to the streams `*out-1*` and `*out-2*` to the file out.results in the results directory. Also it will direct all output to `*out-3*` to the file out3.results in the results directory.

A test may be run with the run-test macro. The results produced from evaluation of the test forms are compared with the compare-results-to-benchmark macro. The syntax and operation of that macro is presented below:

Syntax:

```
(compare-results-to-benchmark test-name
   &key (filter #'identity) (predicate #'equalp)
        (read-function  #'read-line) (quiet  t) verbose)
```

Here is a brief catalog of the other LTM functions:

Defining Tests and Test Suites:
- define-test - establishes the test in LTM
- undefine-test -voids the establishment of the test
- describe-test - display a brief description of the test

identified by a name. Information about the tests are stored in the *test-table*.

3.1. Some Definitions

A *test* is a set of lisp functions and macros that produces some output to a set of streams. The word *stream* is used in the same way as in Common Lisp and is an object that serves as a sink or source for data. Associated with each test are several attributes. The first attribute of a test is its *name* . The *name* is a string or symbol that uniquely identifies a test. Another attribute of a *test* is its *filename*, also called the *test file*. This is the name of the file that contains the series of forms to be tested. Each *test* has a *description* which is a string that explains what the *test* does.

Also associated with each test is a *mapping*. This *mapping* associates with each test a set of output-stream / file pairs. These pairs are used to associate streams of output with files in the following way. All output to the stream part of a pair is redirected to the file identified by the file part of the pair.

Each test may have a *prolog* or *epilog* file associated with it. The prolog specifies the name of a file containing functions and macros to be evaluated before those forms from the test file. The epilog is a file that contains forms to be evaluated after the forms in the test file are evaluated. The difference between the sets of forms contained in the epilog or prolog files and the forms in the test itself is that only the outputs from the functions and macros in the test file are captured by the test. The results file then is *unaffected* by the forms in *either* the epilog or prolog file. The purpose of the forms in the prolog is to set up needed conditions for the test, while the purpose of the forms in the epilog file is to take down conditions from the test. For example, if one wanted to run a test without GC a form in the prologue could do a GC-OFF and a form in the epilog could do a GC-ON. The forms in the epilog are guaranteed to be executed even if the test does not run to conclusion.

This (redirected) output from a test is called the *results* of a test. An instance of the *results* that has been declared correct is called a *benchmark* for the *test*. To *update* a *benchmark* is to deem some *results* correct and to establish these results as the *benchmark*.

Another operation on results is to *compare* them to a *benchmark*. The process of comparing results to a benchmark entails reading from both the benchmark and results file, and comparing the contents of the files. LTM provides a very general way to compare files: The user can specify a filter function (to remove unneeded results such as a dates) as well a comparison function (to define what is meant by equality, e.g. does “2” = “2.0”?). Let us call the items in the *results* file *result forms* and the items in the *benchmark* file *bench forms*. A *filter* is a function to be applied to both the *result* and *bench forms*. and is associated with a *test*.. A *test predicate* is also a *function* that takes two arguments and is also associated with a *test*.. If the *test predicate* returns non nil when given the *filter* applied to the *result form* as its first argument and the filter applied to the *bench form* as its second argument the forms are said to *compare*. The *results* are said to compare if for all *result forms* the corresponding *bench form* compares. A set of tests is called a *test suite* and is

The function *create-machine-dependent-path* takes the same arguments as create-machine-independent-path yet returns a machine dependent path. So to continue our example

```
(CREATE-MACHINE-DEPENDENT-PATH
    :host 'example :directory $data :name 'test)
```

might return something like "spike:spike.develop.data;test.lisp#>" on an Explorer Lisp machine.

We have developed a set of functions to go between machine dependent and machine independent representations with and without doing translations. These are cataloged below:

- machine-independent-path - turn MDP into a MIP without translations
- machine-dependent-path - turn MIP into a MDP without translations
- get-translation-for - interpret as a translation relative to a logical host
- do-translations - map over atoms doing translations relative to a logical host
- show-translations - display translation for one or more logical hosts
- clear-translations - clear the translations table
- directory-p - a predicate that can be applied to both MIP and MDPs
- directory-present - a machine independent way to check for the existence of a directory
- dirs-in-directory - returns all directories contained in a MDP
- directory-files - a recursive directory files with confirmation

3. A Lisp Test Manager

The Lisp Test Manager (LTM) is a portable Lisp regression test package. Regression testing is useful for detecting errors induced by changes made during software maintenance or enhancement (Bucher75). The regression test manager described here can be used for both functional and performance testing as described in (Fairley85).

The general idea for the testing system is to provide a way to systematically test a set of Lisp forms in a portable way. The testing system described here is similar in flavor to Dec Test Manager (DTM on the VAX). A user creates a test by creating a file and putting a series of forms in this file. All output to any number of streams that are referenced by the tested forms are *caught* in a file. This captured output can be stored and compared with other generations of output. The differences between the output generations can then be analyzed. Tests are defined and stored in a table in memory which may be saved (loaded) to (from) disk. Tests may also be grouped into collections called suites.

there is no explicit support for the definition of logicals in Common Lisp itself. In constructing a method for machine independent translation, the basic idea is to have a machine independent way to specify a logical name for a directory path or device. These logicals are then used in conjunction with the machine independent path functions as will be shown latter.

Logical translations are set up in a table that may be populated with a call to to a definition function *define-machine-independent-translation*. These translations are associated with a logical-host. Associated with the logical-host is a physical host and a set of translations for that host for devices and directories. All arguments are put into a canonical form by the function. The complete syntax for the definition function is given below:

```
(DEFINE-MACHINE-INDEPENDENT-TRANSLATION (&key
  (logical-host *SYSTEM-LOGICAL-HOST*)
  (physical-host (where-i-am)) translations device)
```

Our implementation of logicals is constructive and recursive. Multiple calls to define-machine-independent-translation augment the logicals and translations can be defined on top of other logicals. They may be declared in any order as the function uses a topological sort (Kruse87) to decide the order in which logicals must be defined to avoid forward references. Here is an example of the use of the function :

```
(define-machine-independent-translation
   :logical-host 'example
   :physical-host 'spike
   :devices  '(($logical-device  disk$example))
   :translations  '(($data   ($spike data))
                    ($spike   (cerb p1 spike $release))
                    ($release develop)))
```

The function *create-machine-independent-path* takes the components of a MIP and returns a MIP doing the substitution of logicals . Here is the syntax:

```
(CREATE-MACHINE-INDEPENDENT-PATH
      (&key (host *SYSTEM-LOGICAL-HOST*) device directory name
            (type *LISP-FILE-TYPE*) (version :NEWEST))
```

Here is an example of its use using the translations from above:

```
(CREATE-MACHINE-INDEPENDENT-PATH
     :host 'example :directory $data :name '999p)
```

will return the following machine independent path construct:

```
#S(MACHINE-INDEPENDENT-PATH
   :HOST "Spike" :DEVICE NIL :DIRECTORY ("SPIKE" "DEVELOP" "DATA")
   :NAME "999P" :TYPE "LISP" :VERSION  :newest)
```

1. Introduction

This paper presents a set of tools for the development of Common Lisp across a set of machines running different implementations of Common Lisp. The set of tools includes a test manager, a system builder, a code transfer and archival mechanism and a librarian facility. These tools are implemented in Common Lisp (Steele84) and CLOS (DeMichiel89) (Keene89). The usefulness of these tools has been demonstrated in the development of a medium scale software project called SPIKE (Miller88) at the Space Telescope Science Institute.The SPIKE project consisted of some 60K lines of Lisp code. and runs on several Common Lisp implementations including VAXLISP, Allegro Common Lisp, and TI Explorer Common Lisp on a host of different machines.

Although traditionally the software life-cycle for AI software development has been considered to be fundamentally different from that of conventional software, (Partridge86) it has been our experience that a tool set that supports the testing and implementation phases of a software life cycle are just as important in the AI domain as well. Other authors have discussed Lisp based exploratory programming environments (Sheil83) and knowledge engineering environments (Erman88) but we discuss here a more general purpose and machine independent software development tool set.

While it is true that many implementations of Common Lisp have tools to support different phases of the software life cycle, unlike the Ada Programming Support Environment (APSE) in the Ada community (Freedman85), the Common Lisp community does not have a unified view of the tool set. . In this paper we discuss the set of tools we have developed to support our development process across a number of machines and suggest how this set of tools may be extended in the future by integrating them with an intelligent data dictionary.

2. Machine Independent Paths and Logicals

At the heart of the tool set is the ability to represent the location of a file in a machine independent way. A *machine independent path* (MIP) is an abstract data type based upon the Common Lisp notion of a path (Steele84 p 410-418). Like a Common Lisp pathname a machine independent path has components for the host, directory, name, type and version for a file which are accessible with accessor functions. Unlike a pathname a MIP has a standard representation across machines and implementation. Thus a function may reference a file by a MIP and can be shared among machines and implementations without the use of compiler directives. A *machine dependent path* (MDP) is just a namestring in the syntax of the local machine. A MIP is similar in concept to the notion of pathname in APSE and has the the machine independence of the MINT system (Godfrey85) and the portable model of a filing system used by Seismic Kernel System. (Hatton88).

Many implementations of Common Lisp support a notion of logical file names. However

Tools and Techniques for the Development of Common Lisp Applications Across Heterogeneous Machines

Shon Vick
Vick@ STScI.edu
(301) 338-4508

and
Jeff Sponsler

Space Telescope Science Institute
3700 San Martin Dr.
Baltimore, MD 21218

Abstract:

This paper discusses a set of tools and techniques useful in the development of Common Lisp software in a heterogeneous machine environment. The set of tools described include a test manager, a system builder, a code transfer and archival mechanism as well as a librarian facility. These tools are implemented in Common Lisp and the Common Lisp Object System (CLOS) utilizing a machine independent file path construct which is also described. The paper also includes a comparison of these tools with their counterparts in conventional programming environments from a software engineering perspective. Further extensions to the tool set are also proposed.