Massachusetts Institute of Technology
Artificial Intelligence Laboratory

Working Paper 261 August 1984

BUILD -- A System Construction Tool

Richard E. Robbins

Abstract

BUILD is a proposed tool for constructing systems from existing modules. BUILD system descriptions are composed of module declarations and assertions of how modules refer to each other. An extensible library of information about module types and module interaction types is maintained. The library contains information that allows BUILD to derive construction dependencies from the module declarations and referencing patterns enumerated in system descriptions. BUILD will support facilities not adequately provided by existing tools; including automatic derivation of system descriptions, patching of systems, and incorporation of information about how modules change (e.g. the ability to differentiate between the effect of adding a function definition and the effect of adding a comment).

Table of Contents

List of Figures

Introduction

This paper proposes BUILD, a tool for constructing systems from existing modules. A major theme of BUILD is to make system maintenance easier by hiding construction details. The user requests that a system or subsystem be produced and BUILD supplies it. BUILD automatically deduces what subsystems need to be rebuilt, if any, and what command sequence needs to be executed in order to obtain the desired system.

BUILD utilizes system descriptions based on how modules refer to each other in order to derive an internal representation of how modules depend on each other when they are combined to form systems. This mechanism is different from the scheme used by MAKE [Feldman 79] and DEFSYSTEM [Weinreb and Moon 81] that is based on enumeration of construction dependencies. The shift of focus towards module references allows system descriptions to directly reflect design decisions that can only be inferred from construction dependency based descriptions.

BUILD is not constrained to maintaining a limited set of system families. As it becomes necessary to support new kinds of systems, BUILD can be extended to handle them. To achieve this flexibility, BUILD uses an extensible library of information about module types, module interaction types and module construction processes.

The BUILD system description mechanism provides a good foundation for several extensions that existing tools do not support adequately. BUILD will be extended to derive partial system descriptions from source code with a minimum of interaction with the system designer. BUILD will support patching and debugging of large systems by relaxing, but not ignoring, construction constraints. Finally, BUILD will make use of information about how modules change in order to limit the amount of work done to integrate an updated module into a system.

Chapter 1 explores the aspects of large system maintenance that BUILD addresses. Chapter 2 presents several existing system construction tools. Chapter 3 introduces the foundation of the BUILD system - the BUILD system description mechanism. Chapter 4 presents the extensions to BUILD mentioned above. The appendix describes the internal model of module construction dependencies used by BUILD and how assertions about module references cause it to be modified.

1. Maintaining Large Systems

DeRemer and Kron introduced the terms programming-in-the-large and programming-in-the-small [DeRemer and Kron 76] to distinguish between the writing of modules and the structuring of modules into systems. Programming-in-the-small issues include data structures, algorithms and control flow. When writing and maintaining the modules that comprise any system, large or small, problems of programming-in-the-small must be solved. Most programming languages are well equipped for dealing with these issues. When managing large systems the issues of programming-in-the-large must also be dealt with. These issues include module interconnection specification, version control, and consistent compilation in the face of module revision. Most programming languages do not include facilities for programming-in-the-large and therefore need to be used in concert with other tools in order to support the management of large systems.

There needs to be a way for designers to communicate module structure and dependencies to maintainers. For small systems, this can usually be accomplished without any additional effort since system structure can often be determined by examining source code. This is not the case with large systems, as their size and complexity makes comprehending the source code difficult at best. The large system maintainer needs supporting documentation. This documentation can be in many forms. It can be a piece of text, an informal description, or a more formalized description of the system. These descriptions should be easy to prepare and easy to understand.

As systems evolve it is necessary to revise modules and then re-integrate them. When small systems are developed this is usually achieved by recompiling the entire system. This approach is not practical when large systems are being maintained due to the high costs associated with compilation. The ability to separately compile modules, as supported in programming languages like Lisp, C, and CLU is not enough, by itself, to solve this problem. There needs to be a way to identify the modules that might be affected by a change introduced into some other module. It is important that the correct set of modules be rebuilt and relinked into the system as a bug caused by ignoring a module that should be rebuilt can be very difficult to find. This problem, called the consistent compilation problem, becomes more difficult when many people are working together to maintain a system. System construction tools should be able to identify the subsystems that need to be rebuilt when a module changes and they should be able to issue the commands needed to perform the reconstruction without user intervention.

It is very common to produce a version of a system that is known to be incomplete or inconsistent in order to debug a small part of it. When doing this kind of work, system maintainers should not have to be concerned about producing consistent systems. System construction tools should also support this style of maintenance by allowing such changes to be introduced and by providing information about the inconsistencies that may result.

2. System Construction Tools

There are a variety of tools that were designed to support the maintenance of large systems. MAKE, available as part of UNIX[1], is a simple tool for constructing systems that has received widespread use. MAKE uses system descriptions that are composed by enumerating all of the construction dependencies that exist between the system components. A good introduction to MAKE is provided by Feldman [Feldman 79]. DEFSYSTEM, part of the Lisp Machine environment, is a similar tool for managing Lisp programs. Like MAKE, DEFSYSTEM relies on construction dependency based system descriptions. DEFSYSTEM is presented in the Lisp Machine Manual [Weinreb and Moon 81]. There are a series of tools that are driven by system descriptions based on module references instead of construction dependencies. These projects were inspired by the DeRemer and Kron paper and are presented in several papers [Thomas 76, Mitchell 79, Cooprider 79, Tichy 80, Schmidt 82]. The group of languages that evolved from this work are called Modular Interconnection Languages (MIL's). The MIL based tools deal with more than just system construction. They are concerned with issues like version control, management of families of systems, and the development of large software systems in a distributed environment. Since they attack issues that are important but orthogonal to the issues addressed in this project, the MIL based tools will not be presented at this time.

The terminology introduced in this paragraph will be used to refer to the different kinds of modules that are manipulated during the construction process. SOURCE modules are the modules that are produced by people and not programs (e.g. programming language source code). SOURCE modules are manipulated by programs to produce DERIVED modules (e.g. object code). Modules that are the final products of the construction process are called GOAL modules (e.g. executable images of programs). While GOAL modules are usually DERIVED modules, they can also be SOURCE modules. DERIVED modules that are not GOAL modules are called INTERMEDIATE modules (e.g. object code that requires linking in order to form executable images).

[1] UNIX is a trademark of Bell Laboratories

2.1 MAKE

MAKE uses a text file, called a MakeFile, that contains a description of the system to be built. MakeFile entries are of the form:

```
TARGET-MODULE: MODULE1 MODULE2 ... MODULEM
        COMMAND1
        COMMAND2
           ...
        COMMANDN
```

Each entry declares that the target module *depends* on each of the modules to the right of the colon. In MAKE, module B DEPENDS on module A if a change in A implies that B should be reconstructed in order to ensure module consistency. If any of the modules is changed the command sequence below the construction dependency declaration line is executed in order to update the target module. There are no constraints placed on the commands that appear in the command sequence. There are no special ordering rules for MakeFile entries.

MAKE has a simple macro substitution facility. A macro is defined in the following manner:

```
MACRO-NAME=MACRO-EXPANSION
```

All instances of MACRO-NAME enclosed within parentheses and preceded by a dollar sign:

```
$(MACRO-NAME)
```

are replaced by the text of MACRO-EXPANSION when the MakeFile that includes the macro definition is processed.

Construction Algorithm

When MAKE is invoked, a MakeFile is processed and a construction dependency graph is built. Each target module in the MakeFile corresponds to a node in the graph. The modules that a target depends on appear as children of the node that corresponds to the target. A request to "make" a target module is handled by doing a depth-first walk of the graph starting with the node that corresponds to the target. At each node visited, any modules that are missing or older than the modules they depend on are updated. MAKE uses module dating information as an approximate means of noting when changes occur. Since UNIX allows the dating attributes of files to be modified by the user, it is possible to fool MAKE by changing file attributes. However, since most people do not change file attributes, the MAKE mechanism is very reasonable. By rebuilding targets whenever the modules that they depend on are updated, MAKE is guaranteed to perform the correct action when a change that affects a target is made to one of the modules that it depends on. Figure 2-1 contains the MAKE construction algorithm expressed in Lisp.

```
(DEFUN MAKE (NODE)
  (DOLIST (CHILD (GET-CHILDREN NODE))
    (MAKE CHILD))
  (IF (OR (NON-EXISTANTP NODE) (CHILDREN-UPDATEDP NODE))
      (UPDATE NODE)))

(DEFUN GET-CHILDREN (NODE)
  ;; RETURN A LIST CONTAINING THE CHILDREN OF NODE
  )

(DEFUN NON-EXISTANTP (NODE)
  ;; RETURNS TRUE IF AND ONLY IF THE MODULE THAT NODE REPRESENTS
  ;; IS NOT IN THE FILE SYSTEM
  )

(DEFUN CHILDREN-UPDATEDP (NODE)
  ;; RETURNS TRUE IF AND ONLY IF THE MODULE THAT NODE REPRESENTS
  ;; IS OLDER THAN ANY OF THE MODULES THAT IT DEPENDS ON
  )

(DEFUN UDPATE (NODE)
  ;; ISSUE THE COMMAND SEQUENCE ASSOCIATED WITH NODE AS
  ;; SPECIFIED IN THE MAKEFILE
  )
```

Figure 2-1: MAKE Construction Algorithm

A Small Example - TinyComp

TinyComp, a small compiler, is presented in order to demonstrate the basic features of MAKE. The TinyComp example was adapted from one used by Feldman [Feldman 79]. TinyComp has two major components, a parser and a code generator. The parser is built by the YACC parser generating tool. YACC takes a BNF grammar for a language and produces the corresponding parser implemented as a C program. The code generator is implemented directly as a C program, no YACC-like tool is used to derive it.

The two parts use a common set of definitions that describe the shared data structures. These definitions are combined with the source programs at compile time. The compiled programs are loaded with a library that is also subject to change. Figure 2-2 models the construction process for TinyComp. Figure 2-3 contains a MakeFile for TinyComp.

The four entries in figure 2-3 are interpreted in the following manner:

PARSER.C	Depends on PARSER.GRAMMAR. It is updated by running YACC to produce a new parsing program.
PARSER.O	Depends on PARSER.C and DEFINITIONS.C. It is updated by recompiling the parser.
CODEGEN.O	Depends on CODEGEN.C and DEFINITIONS.C. It is updated by recompiling the code generator.
TINYCOMP	Depends on CODEGEN.O, PARSER.O, and LIBRARY.O. It is updated by relinking the system.

MAKE will perform a minimum of work when modules in the TinyComp system change. A change to PARSER.GRAMMAR will cause a new parser to be derived, compiled, and linked. A change to CODEGEN.C will cause CODEGEN.C to be compiled and linked. A change to DEFINITIONS.C will cause PARSER.C and CODEGEN.C to be compiled and linked. A change to LIBRARY.O causes linking but no compiling.

An Extended Example - Lint

The Lint [Johnson 78] system is presented as an extended example of MAKE. Lint examines C source programs and detects bugs that most C compilers cannot. It is also sensitive to constructs that are legal but may not be portable.

Lint consists of a UNIX shell script driver, a set of Lint Library files, and two C programs. Before programs are processed by the first C program (the first pass of Lint), they are processed by the C pre-processor, which handles macro expansion and some compiler directives.

After being pre-processed, programs are sent to the first pass of Lint. It does lexical analysis on the input text, constructs and maintains symbol tables, and builds trees for expressions. An intermediate file that consists of lines of ASCII text is produced. Each line contains an external identifier name, an encoding of the context in which it was seen (use, definition, declaration, etc.), a type specifier, and a source file name and line number. The information about variables local to a function or file is collected by accessing the symbol table, and examining the expression trees. Comments about local problems are produced as detected. The information about external names is collected onto the intermediate file.

Lint libraries are collections of definitions of external names that are appended to the intermediate file generated by the first pass of Lint. They are used to provide Lint with a set of definitions for

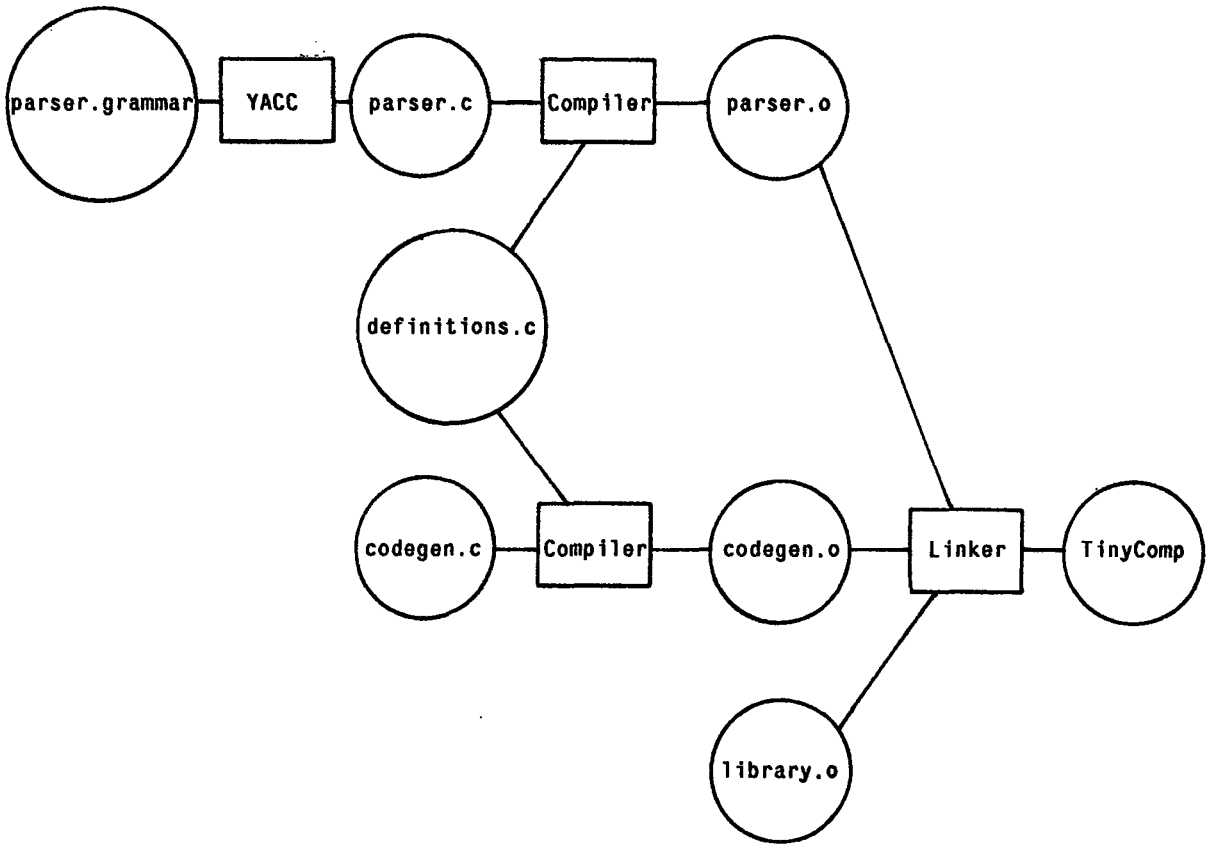

Figure 2-2: Construction Graph For TinyComp

```
PARSER.C: PARSER.GRAMMAR
        YACC PARSER.GRAMMAR      #GENERATE PARSER INTO FILE Y.TAB.C
        MV Y.TAB.C PARSER.C      #CHANGE NAME OF OUTPUT

PARSER.O: PARSER.C DEFINITIONS.C
        CC -C PARSER.C           # -C FOR COMPILATION

CODEGEN.O: CODEGEN.C DEFINITIONS.C
        CC -C CODEGEN.C          # -C FOR COMPILATION

TINYCOMP: CODEGEN.O PARSER.O LIBRARY.O
        CC CODEGEN.O PARSER.O LIBRARY.O -O TINYCOMP # -O FOR LINKING
```

Figure 2-3: MakeFile For TinyComp

commonly used external names without processing the source that contains the definitions. The most commonly used libraries contain the definitions for the functions that are supplied by the UNIX C run time environment. Users can create their own libraries of commonly used names in order to alleviate repeated processing of commonly used stable source modules.

After all the source files and library descriptions have been collected, the intermediate file is sorted to bring all information collected about a given external name together. The second pass of Lint then reads the lines from the intermediate file and compares all of the definitions, declarations, and uses for consistency.

Figure 2-4 contains the MakeFile for the Lint system. The primary point of this example is that MakeFile descriptions, for even medium sized systems like Lint, are very large and difficult to understand. The BUILD description mechanism introduced in chapter 3 provides a much simpler way to describe systems.

The first part of the Lint MakeFile contains macro definitions. These definitions are used to specify directories (e.g. M), compilation flags (e.g. CFLAGS), and to group files (e.g. LINTLIBS). The target `all` is used to name the major subsystems of the Lint system. Since the shell driver requires no construction, it is not listed in the target specification for `all`. The next cluster of specifications manage the first pass of Lint. There is an entry for each library file to be provided with Lint. Each of these specifies that a Lint library file is dependent upon a library source file and the first pass of Lint. Libraries depend on the first pass of Lint because they are constructed by it. The targets that specify management for the second pass of Lint are `lpass2` and `lpass2.o`. The `lintall`, `install`, `shrink`, and `clean` targets are not system modules at all, rather they are used to initiate installation and removal of Lint; a request to "make" any of these will always result in the associated command sequence being executed because they do not exist as modules in the file system. The use of non-existing modules to force command sequences to be executed is a popular and useful feature of MAKE.

MAKE Deficiencies

The problems with MAKE spring from the same source as its strengths - its simplicity. MAKE is a very simple tool; it is easy to learn and use. There are no complicated declaration mechanisms and few constraints are placed on the builder of system descriptions. However, the lack of a mechanism that centralizes construction information forces each command sequence to be completely elaborated. The lack of any constraints on command sequences makes it impossible to determine if a sequence is appropriate or if it has any effect at all on the module represented by the target. Because of its simple

```
M=/usr/src/lib/mip
CFLAGS=-O -DFLEXNAMES
LINTLIBS=llib-port.ln llib-lc.ln llib-lm.ln llib-lmp.ln llib-lcurses.ln

all:    lpass1 lpass2 $(LINTLIBS)

lpass1: cgram.o xdefs.o scan.o comm1.o pftn.o trees.o optim.o lint.o hash.o
        CC cgram.o xdefs.o scan.o comm1.o pftn.o trees.o optim.o lint.o hash.o -o lpass1

trees.o: $(M)/manifest macdefs $(M)/mfile1 $(M)/trees.c
        CC -c $(CFLAGS) -I$(M) -I. $(M)/trees.c
optim.o: $(M)/manifest macdefs $(M)/mfile1 $(M)/optim.c
        CC -c $(CFLAGS) -I$(M) -I. $(M)/optim.c
pftn.o: $(M)/manifest macdefs $(M)/mfile1 $(M)/pftn.c
        CC -c $(CFLAGS) -I$(M) -I. $(M)/pftn.c
lint.o: $(M)/manifest macdefs $(M)/mfile1 lmanifest
        CC -c $(CFLAGS) -I$(M) -I. lint.c
scan.o: $(M)/manifest macdefs $(M)/mfile1 $(M)/scan.c
        CC -c $(CFLAGS) -I$(M) -I. $(M)/scan.c
xdefs.o: $(M)/manifest $(M)/mfile1 macdefs $(M)/xdefs.c
        CC -c $(CFLAGS) -I$(M) -I. $(M)/xdefs.c
comm1.o: $(M)/manifest $(M)/mfile1 $(M)/common macdefs $(M)/comm1.c
        CC -c $(CFLAGS) -I. -I$(M) $(M)/comm1.c
cgram.o: $(M)/manifest $(M)/mfile1 macdefs cgram.c
        CC -c $(CFLAGS) -I$(M) -I. cgram.c

cgram.c: $(M)/cgram.y
        yacc $(M)/cgram.y
        mv y.tab.c cgram.c

llib-port.ln: llib-port lpass1
        -(/lib/cpp -C -Dlint llib-port | ./lpass1 -puv > llib-port.ln )
llib-lm.ln: llib-lm lpass1
        -(/lib/cpp -C -Dlint llib-lm | ./lpass1 -puv > llib-lm.ln )
llib-lmp.ln: llib-lmp lpass1
        -(/lib/cpp -C -Dlint llib-lmp | ./lpass1 -puv > llib-lmp.ln )
llib-lc.ln: llib-lc lpass1
        -(/lib/cpp -C -Dlint llib-lc | ./lpass1 -v > llib-lc.ln )
llib-lcurses.ln: llib-lcurses lpass1
        -(/lib/cpp -C -Dlint llib-lcurses | ./lpass1 -v > llib-lcurses.ln )

lpass2: lpass2.o hash.o
        CC lpass2.o hash.o -o lpass2

lpass2.o: $(M)/manifest lmanifest
        CC $(CFLAGS) -c -I$(M) -I. lpass2.c

lintall:
        lint -hpv -I. -I$(M)  $(M)/cgram.c $(M)/xdefs.c $(M)/scan.c \
                $(M)/pftn.c $(M)/trees.c $(M)/optim.c lint.c

install: all SHELL
        install -s lpass1 /usr/lib/lint/lint1
        install -s lpass2 /usr/lib/lint/lint2
        for i in llib-*; do install -c -m 644 $$i /usr/lib/lint; done
        install -c SHELL /usr/bin/lint

shrink:
        rm -f *.o

clean: shrink
        rm -f lpass1 lpass2 cgram.c $(LINTLIBS)
```

Figure 2-4: MakeFile For Lint

and unconstrained approach, MAKE system descriptions, for all but the smallest of systems, are hard to read and hard to construct.

Construction dependency based system description mechanisms, like the one used by MAKE, produce descriptions that are difficult to understand because systems are not designed with the construction process foremost in mind. Construction dependencies are a byproduct of the ways that modules refer to each other; references imply construction dependencies. For example, the fact that module A contains a call to module B requires the object code of A and B to be linked together in order for a system that includes A to function. MAKE expects that construction dependencies be supplied instead of inferring them from assertions of how modules refer to each other. This would not be such a severe difficulty if MAKE included a facility for deriving descriptions from source code, however, MAKE does not have such a facility.

MAKE does not include an adequate means for saving and reusing common construction patterns. The introduction of such a facility would allow system descriptions to be shorter since several similar elaborations would be replaced with a single identifier. The definition of the identifier would document and highlight the construction pattern. The MAKE macro facility is too simple; it does not even allow for parameterized macros.

Maintainers can only effect changes to a system by manipulating source modules or requesting goal modules; the use of intermediate modules in system descriptions only serves to make them longer and more complex. However, since intermediate modules play an important role in the construction process, it is very natural for MAKE descriptions to include references to them. The Lint MakeFile is filled with references to C object code modules. The need to reference intermediate modules in MAKE system descriptions is a shortcoming of MAKE.

MAKE allows system descriptions to omit source modules that are also goal modules since there is no command sequence that uses or effects them. For example, there is nothing that forces the UNIX Shell Script that drives Lint to be included in the MAKE description of Lint. This would be a serious omission if someone were using the Lint MakeFile in order to determine which modules should be copied in order to transport Lint.

2.2 DEFSYSTEM

DEFSYSTEM is used to install and maintain Lisp Machine software systems. It is similar to MAKE in that the user is expected to supply a description of how each module in the system is constructed and what the construction dependencies are. However, it differs from MAKE in several significant ways. DEFSYSTEM maintains installed versions of software, it is not designed to store systems in files. This difference is due to differences in the computing environments of Lisp and UNIX. In UNIX, systems are stored in the file system and loaded when they are called. In the Lisp environment that DEFSYSTEM was designed to operate in, systems are loaded into the environment when the Lisp environment is booted. DEFSYSTEM system descriptions are more constrained than MAKE descriptions. When using DEFSYSTEM, construction sequences, called transformations, must be formally defined before they are used. This is different from the MAKE approach of allowing unlimited use of UNIX command sequences.

System descriptions are made by calling the **defsystem** function. Calls have the form:

*(**defsystem** system-name (keyword args...) (keyword args...) ...)*

The options selected by the keywords fall into two general categories: properties of the system and transformations.

Here is a partial listing of DEFSYSTEM property keywords:

:name Specifies a "pretty" version of the name for the system for use in printing.

:module Assigns a name to a group of files within the system.

A transformation is an operation, such as compiling or loading, that takes one or more files and performs some operation on them. DEFSYSTEM transformations are of two types: simple and complex. A simple transformation is a single operation on a module, such as compiling it or loading it. A complex transformation combines several transformations; for example, loading the results of a compilation. The general format of a simple transformation is:

(transformation-name input pre-conditions)

transformation-name
The name of the transformation to be performed on the files specified by *input*. Examples of transformation names are **:fasload** and **:compile-load-init**.

input A module or nested transformation.

pre-conditions Optional. Specifies transformations that must occur before the current transformation itself can take place. The format is either a list *(transformation-name module-names ...)*, or a list of such lists. Each of these lists

declares that the transformation *transformation-name* must be performed on the indicated modules before the current transformation can take place.

It is important to distinguish between transformation declarations and transformation references. Transformations are declared as part of the keyword list in calls to **defsystem**. Transformations are referenced in pre-condition[2] lists. The transformations referenced in a pre-condition list must be declared somewhere in the DEFSYSTEM description.

The following simple transformations are pre-defined:

:fasload Loads the indicated file when a newer version of the file exists than was read into the current environment.

:compile Compiles the indicated file when the source file has been been updated since the compiled code file was written.

Unlike simple transformations, complex transformations do not have any standard form. The pre-defined complex transformations are:

:compile-load Compiles and then loads the input files. It has the form: *(***:compile-load** *input compile-pre-conditions load-pre-conditions)* and is exactly the same as *(***:fasload** *(***:compile** *input compile-pre-conditions) load-pre-conditions)*. Everything after *input* is optional.

:compile-load-init
Compiles and loads the input files. This transformation is sensitive to changes made to an additional dependency list. It has the form: *(***compile-load-init** *input additional-dependencies compile-pre-conditions load-pre-conditions)*. Everything after *additional-dependencies* is optional. The input module will be compiled and loaded whenever the source file for *input* or any of the modules listed in *additional-dependencies* is updated.

Unlike the other transformations, the additional-dependencies section of the **:compile-load-init** transformation specifies the same kind of construction dependency as MakeFile entries do.

DEFSYSTEM contains a facility for defining new transformations. New simple transformations are defined using the **define-simple-transformation** function. The function has the form:

*(***define-simple-transformation** *name function default-condition input-file-types output-file-types)*

name The name of the transformation being defined.

function The Lisp function to be called when the transformation is performed.

default-condition The function that is called in order to determine if the transformation should be performed.

---

[2]Lisp Machine documentation calls *pre-conditions dependencies*.

input-file-types Specifies the types of the input files to the transformation. Type specifications are just file name extensions (e.g. .LISP or .BIN).

output-file-types Specifies the types of the output files produced by the transformation.

For example, to define a simple transformation called :ZCompile that calls the ZComp function to transform Z source files into binary files the following definition should be made:

```
(DEFINE-SIMPLE-TRANSFORMATION :ZCOMPILE ZCOMP
        FILE-NEWER-THAN-FILE-P (:Z) (:BIN))
```

The compiler will be invoked whenever the input file (Z source) is newer than the output file (binary file). In other words, the transformation will be performed whenever the source file is updated.

Complex transformations are defined as Lisp macros. Here is the definition of the **:compile-load** transformation that was described earlier:

```
(DEFMACRO (:COMPILE-LOAD DEFSYSTEM-MACRO)
  (INPUT &OPTIONAL COMPILE-PRE-CONDITIONS LOAD-PRE-CONDITIONS)
  '(:FASLOAD (:COMPILE ,INPUT ,COMPILE-PRE-CONDITIONS)
             ,LOAD-PRE-CONDITIONS))
```

Construction Algorithm

Systems previously modeled with the DEFSYSTEM function are constructed by calling the **make-system** function. Calls have the form:

*(**make-system** system-name)*

The construction dependency graph specified by the transformations and pre-conditions in the DEFSYSTEM model of the system *system-name* is analyzed in order to determine what construction needs to be done. Each transformation is applied in the following manner:

1. All transformations referenced as pre-conditions are applied.

2. The input module is processed by the construction function if it or any modules listed in additional dependency lists have been updated.

The pre-condition lists order the transformation applications. Like MAKE, DEFSYSTEM uses simple functions based on dating information in order to determine when a module should be reconstructed. However, unlike MAKE, DEFSYSTEM allows the optional specification of predicates that control when construction is done. The new predicates can replace the simple ones that are supplied with DEFSYSTEM.

DEFSYSTEM includes a patch facility. It allows small changes to be made to a system without invoking the DEFSYSTEM transformation/dependency mechanism. Each set of changes is stored in a patch file that typically contains new function definitions or redefinitions of old functions. Each patch is

assigned a number. If a system contains patches, then the patches are loaded, in order, after the unpatched version of the system is loaded.

A Small Example - TinyComp

Figure 2-5 contains the DEFSYSTEM description for a Lisp implementation of the TinyComp system. It is assumed that a **:yacc** transformation that invokes YACC when grammar files are updated has been defined. (If this compiler were really written in Lisp, the parser would be generated using a macro expansion and not a parser generating program like YACC.)

```
(DEFSYSTEM TINY-COMP
  (:MODULE DEFS "DEFINITIONS")
  (:MODULE GRAMMAR "PARSER.Y")
  (:MODULE CODE-GENERATOR "CODEGEN.LISP")
  (:MODULE LIBRARY "LIBRARY")

  (:FASLOAD DEFS)
  (:FASLOAD LIBRARY)
  (:COMPILE-LOAD-INIT CODE-GENERATOR  (DEFS) (:FASLOAD DEFS))
  (:COMPILE-LOAD-INIT (:YACC GRAMMAR) (DEFS) (:FASLOAD DEFS)))
```

Figure 2-5: DEFSYSTEM Description For TinyComp

The TinyComp DEFSYSTEM definition is a set of module definitions followed by a series of transformations. There are four transformations in the definition, they have the following interpretation:

`(:FASLOAD DEFS)`
: Specifies that DEFS should be loaded whenever it is updated. There are no pre-conditions to be satisfied before the loading can take place.

`(:FASLOAD LIBRARY)`
: Specifies that LIBRARY should be loaded whenever it is updated. There are no pre-conditions to be satisfied before the loading can take place.

`(:COMPILE-LOAD-INIT CODE-GENERATOR (DEFS) (:FASLOAD DEFS))`
: Specifies that CODE-GENERATOR should be be compiled and loaded whenever it or DEFS changes. Before the compilation can take place, DEFS must be loaded.

`(:COMPILE-LOAD-INIT (:YACC GRAMMAR) (DEFS) (:FASLOAD DEFS))`
: Specifies that a parser derived by YACC from GRAMMAR is to be compiled and loaded. YACC is invoked to produce a new parser whenever GRAMMAR changes. The compiler and loader are invoked whenever DEFS or the YACC parser changes. YACC will not be invoked if only DEFS changes. Prior to compilation, DEFS must be loaded.

An Extended Example - Lint

A Lisp implementation of Lint is considered in this example. The DEFSYSTEM description, presented in figure 2-6, is much easier to understand than the MAKE description from figure 2-4. The **:build-lint-library** transformation is assumed to have been defined. It has the form:

(:**build-lint-library** *input pre-conditions)*

It constructs Lint library files from Lint library sources. The transformation allows the optional specification of pre-conditions and is applied if either the input module or first pass of Lint are updated.

```
(DEFSYSTEM LINT
  (:NAME "Lint")
  (:MODULE DEFINITIONS-1 ("MACDEFS" "MANIFEST" "MFILE1"  "LMANIFEST"))
  (:MODULE DEFINITIONS-2 ("MANIFEST" "LMANIFEST"))
  (:MODULE PASS1 ("XDEFS.LISP" "SCAN.LISP" "COMM1.LISP" "PFTN.LISP"
                  "TREES.LISP" "OPTIM.LISP" "LINT.LISP" "HASH.LISP"))
  (:MODULE PASS2 ("LPASS2.LISP" "HASH.LISP"))
  (:MODULE DRIVER "SHELL.LISP")
  (:MODULE GRAMMAR "LISPGRAM.LISP")
  (:MODULE LIBRARIES ("LLIB-PORT.LN" "LLIB-LC.LN" "LLIB-LM.LN"
                      "LLIB-LMP.LN" "LLIB-LCURSES.LN"))

  (:FASLOAD DEFINITIONS-1)
  (:FASLOAD DEFINITIONS-2)
  (:COMPILE-LOAD DRIVER)
  (:COMPILE-LOAD-INIT PASS1 (DEFINITIONS-1) (:FASLOAD DEFINITIONS-1))
  (:COMPILE-LOAD-INIT PASS2 (DEFINITIONS-2) (:FASLOAD DEFINITIONS-2))
  (:COMPILE-LOAD-INIT (:YACC GRAMMAR) (DEFINITIONS-1)
                      (:FASLOAD DEFINITIONS-1))
  (:BUILD-LINT-LIBRARY LIBRARIES (:FASLOAD DRIVER GRAMMAR PASS1)))
```

Figure 2-6: DEFSYSTEM Description For Lint

The DEFSYSTEM definition for Lint specifies that the following requirements are placed on the installation of a complete, consistent system:

* All of the files enumerated in the DEFINITIONS-1 module must be loaded.

* All of the files enumerated in the DEFINITIONS-2 module must be loaded.

* The DRIVER module must be compiled and loaded.

* All of the files in the PASS-1 module must be compiled and loaded. The compilation cannot take place until after the DEFINITIONS-1 module has been loaded. A change to the DEFINITIONS-1 module will cause PASS-1 to be compiled and loaded.

* All of the files in the PASS-2 module must be compiled and loaded. The compilation cannot take place until after the DEFINITIONS-2 module has been loaded. A change to the DEFINITIONS-2 module will cause PASS-2 to be compiled and loaded.

* The GRAMMAR module must be processed by YACC, compiled, and loaded. The compilation cannot take place until after DEFINITIONS-1 has been loaded. A change to DEFINITIONS-1 will cause GRAMMAR to be reprocessed.

* The libraries enumerated in LIBRARIES must be built from their source modules. The construction cannot take place until after the DRIVER, GRAMMAR, and PASS-1 modules have been loaded.

DEFSYSTEM Deficiencies

The major problem with DEFSYSTEM is that it requires that systems be described in terms of construction dependencies. This causes system descriptions to be difficult to construct and difficult to understand. DEFSYSTEM provides a higher level language for describing systems than MAKE does; because of this, DEFSYSTEM descriptions are more formalized than their MAKE analogs. The more formal mechanism employed by DEFSYSTEM goes a long way towards solving some of the problems associated with MAKE. However, DEFSYSTEM does not escape the problems that it shares with MAKE and any other construction dependency based system.

DEFSYSTEM transformations must be formally defined. All construction sequences are encapsulated and presented to the system specifier in a uniform fashion. The transformation mechanism incorporates the Lisp macro mechanism. These factors make it easier to produce and understand DEFSYSTEM descriptions than their MAKE counterparts. However, there are no constraints placed on the macro writer. DEFSYSTEM does not perform any checking to ensure that what is placed in a complex transformation macro is a valid transformation.

DEFSYSTEM does not differentiate between source, intermediate, and goal modules. In general, intermediate modules are hidden by complex transformations. For example, there are no references to intermediate modules in figures 2-5 and 2-6. DEFSYSTEM does not force intermediate modules to be included, it does not prohibit them either.

3. BUILD

BUILD is designed to address the problems outlined in chapters 1 and 2. Like the tools mentioned in chapter 2, BUILD uses system descriptions in order to construct internal models of systems in terms of construction dependencies. BUILD minimizes the amount of processing done in order to integrate revised modules into systems by analyzing construction dependency graphs in a manner similar to MAKE and DEFSYSTEM. However, unlike MAKE and DEFSYSTEM, BUILD hides construction dependencies from designers and maintainers. This is accomplished by basing system descriptions on module references instead of construction dependencies.

3.1 Module Declaration Assertions

Most programming environments construct systems by manipulating files, BUILD system descriptions refer to modules. The module declaration assertion bridges the gap between these two views of system components. The assertion provides for the definition of modules as groups of files. All references to a module are actually references to each of the files that comprise the module. Modules and files have types. All files in a module must be of the same type as the module. File type information is inferred from the module type specification field of module declaration assertions. Module declaration assertions have the form:

(module *module-name module-type file-list)*

module-name	The name of the module. Must be unique within the system description.
module-type	The type of the module. The *module-type* must be defined in the library of types. The library of types is described in section 3.4.
file-list	The names of the files that comprise the module.

System modules can be defined as the major abstract components of a system (an example of this is the Lint system description in section 3.3), they can represent the files that comprise a system, they can represent the functions that comprise a system (if the programming environment is not designed to manipulate functions, then each function will have to be in its own file), or any level of abstraction in between.

3.2 Reference Assertions

Reference assertions provide for the specification of references between modules. Reference assertions have the form:

(reference-type referencing-modules referenced-modules)

reference-type The reference that applies between the *referencing* and *referenced* modules. There must be a definition for this reference type with module types that match the types of the referencing module and referenced modules in the library of types.

referencing-modules
A module name or a list of module names. Each module name must be the name of a module as specified in a module declaration assertion.

referenced-modules
A module name or list of module names. Each module name must be the name of a module as specified in a module declaration assertion.

BUILD infers construction dependencies from reference assertions by taking advantage of the fact that construction dependencies are caused by references between modules. If two modules do not refer to each other then it is impossible for there to be a construction dependency that involves them. Some kinds of references cause construction dependencies and others don't. For example, the following assertion,

(Script-Refs *Driver Pass1)*

which specifies that *Driver*, a UNIX shell script, contains a call to *Pass1*, an executable image, does not imply a construction dependency since UNIX shell scripts are interpreted at run time and there is no change to either *Driver* or *Pass1* that can require any construction to be done. However, the following assertion,

(Uses-Definitions-From *Parser CommonDefs)*

which specifies that *Parser* relies on definitions from *CommonDefs*, implies that a change to *CommonDefs* should result in the recompilation of *Parser*.

Here are some reference assertions and the construction dependencies that they imply:

(Includes *A B)* Asserts that *A* contains the contents of *B*. Many C and UNIX tools allow source code to contain references to modules that causes the contents of the referenced module to be placed in the referring module when that module is processed by the tool. With this mechanism it is possible to isolate and share arbitrary text for use in C programs, YACC grammars, etc. The **Includes** assertion implies that whenever the included module, *B*, changes, the including module, *A*, needs to be rebuilt.

*(**Uses-Definitions-From** A B)*

Asserts that *A* relies on definitions (e.g. macros) supplied by *B* and therefore *B* must be loaded in order for *A* to compile properly.

*(**Calls** A B)*

Asserts that *A* contains function calls to *B* and therefore *B* must be loaded into any system that includes *A*.

3.3 BUILD System Description Syntax

The general form of a BUILD system description is:

***(defbuild-description** system-name subsystems assertions)*

system-name The name of the system being described. It does not have to be the name of a module in the system. Requesting that *system-name* be constructed causes all of the *assertions* in the description to be taken into account.

subsystems A list of goal module declarations. Each goal module declaration has the form ***(subsystem** goal-module-name component-modules)*.

assertions A list of module declaration assertions and reference assertions.

The *subsystems* declaration section allows for the existence of goal modules other than *system-name*. Construction of any of the goal modules declared in *subsystems* may be requested. If any of these modules are requested, BUILD will ignore assertions that do not effect the requested module.

A Small Example - TinyComp

Figure 3-1 contains the BUILD system description for a Lisp implementation of TinyComp.

```
(DEFBUILD-DESCRIPTION  TINYCOMP ()

 (MODULE DEFS LISP ("DEFINITIONS"))
 (MODULE PARSER GRAMMAR ("PARSER.GRAMMAR"))
 (MODULE GENERATOR LISP ("CODEGEN.SOURCE"))
 (MODULE LIBRARY  LISP ("LIBRARY" LISP))

 (USES-DEFINITIONS-FROM (PARSER GENERATOR) DEFS)
 (CALLS PARSER (LIBRARY GENERATOR))
 (CALLS GENERATOR LIBRARY))
```

Figure 3-1: BUILD Description For TinyComp

There are two different kinds of reference assertions used in the TinyComp description:

* The USES-DEFINITIONS-FROM assertion specifies that the elements of DEFS need to be loaded before GENERATOR and PARSER can be compiled.

* The CALLS assertions specify that compiled versions of PARSER, GENERATOR, and LIBRARY need to be loaded.

An Extended Example - LINT

Figure 3-2 contains a BUILD system description for a Lisp implementation of Lint. A request to construct Lint will cause all of the information given in the description to be considered. A request to construct one of the subsystems (e.g. LIBRARIES, PASS1, or PASS2) will cause information that does

not affect the other subsystems to be considered. For example, a request to construct PASS1 would cause BUILD to ignore the assertion that LINT2 references HASH.

```
(DEFBUILD-DESCRIPTION  LINT
  ((SUBSYSTEM LIBRARIES (LIBRARY-SOURCES))
   (SUBSYSTEM PASS1 (COMMON-DEFS DEFS-1 LINT1 SCANNER PARSER OPTIMIZER
                     TREES HASH LOW-LEVEL))
   (SUBSYSTEM PASS2 (COMMON-DEFS LINT2 HASH)))

  (MODULE DRIVER LISP ("SHELL"))
  (MODULE COMMON-DEFS LISP ("MANIFEST" "LMANIFEST"))
  (MODULE DEFS-1 LISP ("MACDEFS" "MFILE1"))
  (MODULE LINT1 LISP ("LINT.LISP"))
  (MODULE LINT2 LISP ("LPASS2.LISP"))
  (MODULE SCANNER LISP ("SCAN.LISP"))
  (MODULE PARSER GRAMMAR ("CGRAM.Y"))
  (MODULE OPTIMIZER LISP ("OPTIM.LISP"))
  (MODULE TREES LISP ("TREES.LISP"))
  (MODULE HASH LISP ("HASH.LISP"))
  (MODULE LOW-LEVEL LISP ("PFTN.LISP" "XDEFS.LISP" "COMM1.LISP"))
  (MODULE LIBRARY-SOURCES LINT-LIBRARY-SOURCE
          ("LLIB-LC.LN" "LLIB-LM.LN" "LLIB-LCURSES.LN" "LLIB-PORT.LN"
                        "LLIB-LMP.LN"))

  (USES-DEFINITIONS-FROM (LOW-LEVEL LINT2 HASH TREES) COMMON-DEFS)
  (USES-DEFINITIONS-FROM (LINT1 SCANNER PARSER OPTIMIZER) DEFS-1)
  (CALLS DRIVER (LINT1 LINT2 LIBRARIES))
  (CALLS LINT1 (SCANNER PARSER OPTIMIZER))
  (CALLS SCANNER LOW-LEVEL)
  (CALLS PARSER (TREES HASH LOW-LEVEL))
  (CALLS OPTIMIZER TREES)
  (CALLS (TREES HASH) LOW-LEVEL)
  (CALLS LINT2 HASH)
```

Figure 3-2: BUILD Description For Lint

The Lint BUILD description illustrates the use of *subsystems* and the use of module declaration assertions that contain more than one to one mappings between files and modules. The description declares that the entire system, either of the two passes, or the libraries may be requested. In the example, the system is defined in terms of its major components (i.e. PARSER, COMMON-DEFS, LOW-LEVEL etc.). It is also possible to define Lint solely in terms of the files that comprise it. In this case there would need to be a module definition for each file in the system. Finally, it is possible to define Lint in terms of the major functions that comprise it. In this case each function would need to reside in its own file as the smallest unit manipulated by the Lisp Machine is the file.

3.4 Library Of Types

The Library of Types contains the definition for each assertion. Functions that add new assertion definitions are provided. The function **define-module-type** is used to specify a new module-type for use in module declaration assertions. The function **define-reference-type** is used to specify a new kind of reference assertion.

Defining Module Types

Calls to **define-module-type** are of the form:

(define-module-type *name intermediate instantiation-processing)*

name — The name of the module type (e.g. C-Source, Ada-Specification, C-Parser-Grammar).

intermediate — If this field is T then the module-type being defined is an intermediate module type. Intermediate modules are not allowed in system descriptions but they are used in the internal construction dependency graphs.

instantiation-processing — Specifies construction dependency graph manipulations to be performed when a module of type *name* is instantiated. For instance, when a module that contains a YACC grammar is used for the first time in a reference assertion, the construction dependency graph should be updated to include a node for the grammar module and the fact that YACC should be invoked upon it in order to produce a source code module.

Defining Reference Types

Calls to **define-reference-type** are of the form:

(define-reference-type *name referencing-type referenced-type processing)*

name — The name of the reference being defined (e.g. **Calls, Uses-Definitions-From**).

referencing-type — The *referencing-module* type specification.

referenced-type — The *referenced-module* type specification.

processing — Code to manipulate the construction dependency graph whenever this assertion is used. For instance, the *processing* section for the **Uses-Definitions-From** assertion would cause the construction dependency graph to be updated to model the fact that all elements of the referenced module need to be loaded before any elements of the referencing module can be compiled. This section is similar to the *instantiation-processing* section of the **define-module-type** function.

The type specification of the modules involved in assertions allows BUILD to differentiate between two assertions with the same name but different module types. For example, BUILD can differentiate

between a **Calls** assertion that involves two Lisp modules and a **Calls** assertion that involves two C modules. The type specifications are also used to ensure that assertions are correctly typed. This is similar to the use of interface specifications in strongly typed programming languages.

Eventually, there will be a language for manipulating BUILD construction dependency graphs, but the requirements for this language are not clear yet. Initial implementations of BUILD have instantiation processing sections written in what ever language BUILD is implemented in. The first implementation uses Lisp. As the requirements for a construction graph manipulation language evolve, the means for specifying instantiation processing will be formally specified. The appendix contains a description of the kinds of manipulations that will be necessary.

4. Extensions

This chapter presents extensions to BUILD that will allow it to provide a set of facilities that other tools do not. The extensions take advantage of the BUILD specification mechanism described in chapter 3. The extensions are automatic derivation of system specifications from source code, support for patching and similar maintenance styles, and the incorporation of the nature of module change into the reconstruction algorithms.

4.1 Automatic Derivation of System Descriptions

The BUILD description mechanism provides a natural way to describe systems but it does not ensure that the descriptions are complete or correct. The designer is still required to generate a description of the system by hand. A tool that derives system descriptions from source code will relieve designers from the chore of building system description files. In addition, such a tool should allow the manual construction of system description files so that descriptions can be built for systems that are specified but not yet implemented.

Initial work with BUILD indicates that tools to derive most of the BUILD specifications for C and Lisp systems will not be difficult to construct. In C, the two common references that have an impact on construction dependency analysis are UNIX inclusion, and non-local function calls. Both of these references are relatively easy to identify. In Lisp, dependencies that arise from most function calls and definition requirements are not difficult to identify, however, dependencies that arise from the use of functions as parameters and explicit calls to the Lisp evaluator are difficult to identify. Simple analysis of source code will allow BUILD to infer a great deal about systems. An automatic specification tool will significantly reduce the amount of information that needs to be provided by specifiers and will provide useful system documentation.

The first implementation of BUILD will include a facility for deriving partial specifications for Lisp systems.

4.2 Patching

There are many instances where a system maintainer may want to introduce changes into a system without making sure that the resulting system is consistent. Consider debugging experiments where small changes are introduced to examine some small part of the system. These changes may not be intended to become part of a released system, it may even be known that they will cause compilation of some other module to fail. Another instance where the ability to patch a system is important is when a quick fix is being attempted and it is important that the effects be seen quickly. This kind of change represents a tentative guess on the part of the maintainer. The introduction of such changes into systems must be supported by system construction tools if such tools are going to help and not hinder maintainers.

At the basis of a patching system is the ability to incrementally link modules into existing systems. If this facility is not present then patching cannot be supported. Some computing environments support incremental linking but do so in a manner that is very difficult to use. An unpleasant incremental linking mechanism is not much better than the lack of one because it means that most users will not take advantage of it. BUILD can provide a more pleasant interface to an incremental linking mechanism that is difficult to use.

The DEFSYSTEM patch facility provides some support for producing inconsistent systems. Unfortunately, the DEFSYSTEM patching facility makes no use of the dependency information that the rest of the tool uses. No analysis of the effect of a patch is available. Nothing guarantees that a patch will even be loaded correctly according to the dependency information that is available. For example, if a patch file includes a modified macro definition and two calls to it, the calls will not refer to the new version of the macro unless they are placed after the definition in the patch file by the user.

The first implementation of BUILD will support patching of Lisp systems above and beyond what is supplied by DEFSYSTEM. Unlike the DEFSYSTEM patching mechanism, the BUILD patching mechanism will make use of the construction dependency information while applying patches. In general, the dependency information will be used to ensure that modifications to modules are made in the same order that would result if the system were being constructed instead of patched. The BUILD patching mechanism will also supply information about the effect that a patch may have on the rest of the system. The analysis will be done by propagating the effects of a change through the internal model of the system and then identifying those modules that were affected by the change but ignored by the patch.

4.3 More Precise Change Analysis

All of the tools mentioned in chapter 2 are sensitive to the fact that some change has occurred to a module in a system. However, no attention is paid to the nature of the change. By exploring the nature of a change it is possible to limit the amount of processing done when updating systems.

If source code is changed in a way that cannot alter its compilation, there is no reason for the source module to be recompiled. For example, compilation should not be done when source code has only been reformatted or had commentary added to it. If a function is added to a module, but no existing modules are updated to contain calls to the new function, nothing should be done to the existing modules. Lint libraries are dependent upon the first pass of Lint, however, most changes to the first pass of Lint will not affect the libraries. MAKE and DEFSYSTEM will rebuild the Lint Libraries whenever the first pass of Lint is changed; this unnecessary processing should be avoided.

Change analysis can also provide important debugging information. For example, if a module interface is changed, but not all of the modules that contain references to that module are changed, there is a strong possibility that an error of omission has been made.

Unlike MAKE, DEFSYSTEM can be extended to include more complicated predicates for deciding when changes are significant. There is nothing preventing a DEFSYSTEM system definition from using parsers and source code comparison programs in order to decide when transformations should take place. However, no enhanced predicates are supplied with DEFSYSTEM and none of the DEFSYSTEM descriptions encountered in the process of preparing this paper included definitions of such specialized predicates.

Specialized predicates are only useful when they require less processing to determine that a transformation can be avoided than applying the transformation in the first place. BUILD steps around this issue by assuming that it is a single tool in an integrated environment in which the tools that are used to modify modules can supply information to BUILD about the nature of changes. BUILD provides an interface for communicating information about changes to modules. The library of assertions will be extended to include definitions for *change assertions*; each change assertion is of the form:

(assertion-name modified-module)

For example, the assertion:

(added-struct *defs)*

informs BUILD that *defs* has been changed by adding a new structure and therefore modules that rely on *defs* do not have to be re-compiled. The compilation of unaltered modules can be avoided since there is no way for them to refer to the new structure. The assertions:

(added-comment *defs)*
(re-formatted *defs)*

imply that no changes that can alter the compilation of *defs* have been made and therefore no re-compilation needs to be done.

Change assertions will be defined and stored in the library with a mechanism similar to the one used to specify module declaration and reference assertions.

I. Appendix

BUILD uses two representations of systems. The user visible representation that has been discussed in the body of this paper is called the *reference representation*. The internal representation, called the *construction representation*, is a construction dependency graph similar to the kind used by MAKE and DEFSYSTEM. The construction representation is derived from the reference representation. This appendix describes the construction representation and outlines how it is manipulated when reference assertions are made.

I.1 The Construction Representation

Internally, systems are represented as acyclic directed graphs. There are two kinds of nodes in these graphs, *process* nodes and *module* nodes. Module nodes are used to represent all of the files used in the construction process (e.g. source, goal, and intermediate). Process nodes are used to represent the processes used to produce derived modules. A process depends upon a module if and only if a change to the module could result in a change to the output of the process. System construction dependency graphs maintain the following invariants:

1. The parent of a module node, if there is one, must be a process node. The process node represents the process used to create the file that the module node represents.

2. A module node can have no more than one parent.

3. A module node without a parent corresponds to a source module.

4. The children of a module node, if there are any, must be process nodes. These nodes represent the processes that depend upon the file represented by the module node.

5. A module node without children corresponds to a goal module.

6. The children of a process node must be module nodes. These module nodes represent the derived modules produced by the process represented by the process node. Each process node will have at least one child.

In other words, construction dependency graphs begin with module nodes that represent all of the source modules in a system and end with module nodes that represent all of the goal modules. Module nodes are separated by process nodes that represent the processes that derive later modules from earlier ones. Figure 2-2 is an example of a well formed construction dependency graph.

Process Type Definitions

Process Type Definitions are used to specify the various construction processes that may be invoked when a system is built. Examples of processes that will require definition for C and Lisp systems include C-Compilation, Lisp-Loading, and C-Include-Processing. Process Type Definitions have the form:

(define-process-type *name input-specification output-specification construction-rules)*

The arguments to this function have the following meaning:

name Name of the Process Type being defined (e.g. C-Compilation, Lisp-Loading, and C-Include-Processing).

input-specification Specifies the input modules and types to the process. Input specifications are of the form:

((role$_1$ module-type$_1$) (role$_2$ module-type$_2$) ... (role$_n$ module-type$_n$)).

Roles are used as formal parameters within Process Type Definitions.

output-specification

Lists the formal names and types of the files that are derived by instances of the Process Type. Output specifications are of the form:

((formal$_1$ module-type$_1$) (formal$_2$ module-type$_2$) ... (formal$_n$ module-type$_n$))

construction-rules Specify how the output modules of the process are constructed. Each formal in the output specification must appear exactly once in the construction rules section. Rules are of the form:

((formal-list$_1$ construction-rules$_1$) (formal-list$_2$ construction-rules$_2$) ... (formal-list$_m$ construction-rules$_m$))

Modules connected to the process are referenced by enclosing the desired role name or formal output name within brackets in the construction rules. The command sequence for a formal may be null.

In many situations lists of modules are used to fulfill a single role. For instance, several modules may need to be loaded into the LISP environment prior to compilation or several files may be inserted into the primary file by the C-Include Processor. Rather than specifying many similar processes, i.e., C-Include-1-module, C-Include-2-modules and so on, modules may be merged at the inputs of processes. Within the body of the process type definition, a reference to the role that has received a merged input will be interpreted to apply to each element of the merged list.

The use of null command sequences accommodates the combination of logically separate processes by system utilities. Examples of processes that are often combined with other processes are compilation pre-processing routines like macro-expansion. Although macro-expansion is a logically

separate process from compilation, it is sometimes accomplished by the compiler and is not available separately. Although BUILD models the macro-expansion process apart from compilation there may be no way to accomplish macro-expansion without compilation. In this instance the construction rule for building the macro-expanded module will be null, and the compiler process that will be attached to the macro-expansion process by the macro-expanded module will be invoked to build the proper module.

Process and Module Instances

Each process will have the following attributes:

Internal Name Used by BUILD, not visible to the user.

Type A Process Type as specified in a Process Type Definition.

Input List A list of the modules that are used to fulfill each role as specified in the Process Type Definition for the process.

Output List A list of the modules derived by this process.

Each module will have the following attributes:

External Name The user supplied name for a module. Examples include: lint.c, main.clu and so on. This field is optional. Intermediate modules are not user visible and they do not have external names. Only source and goal modules have external names.

Internal Name Used by BUILD, not visible to the user.

Type A Module Type as specified in a Module Type Definition.

Creator Process The process that derived this module. This field is null for source modules.

Construction Usage List
A list of references to the processes that depend on this module.

I.2 Dependency Graph Manipulations

This section contains a brief description of the kinds of construction dependency graph manipulations that will need to be provided in a language used by reference and module declaration assertion definitions.

Instantiation Functions

These functions will allow for the creation of construction dependency graph nodes and specification of the attributes described in the previous section. They will have the form:

*(***make-process-node** *type input-list output-list)*
*(***make-module-node** *external-name type creator-process usage-list)*

Observation Functions

These functions will allow for the inspection of nodes in the construction dependency graph. In general, given a node and attribute, these functions will return the specified field of the node in question. They will have the form:

*(***get-attribute** *attribute-name node)*

Node Mutation Functions

These functions will allow for changes to be made to the construction dependency graph. Changes to the graph are accomplished by changing the input and output lists of process nodes and the creator process and construction usage lists of module nodes. They will have the form:

*(***set-attribute** *attribute-name node new-contents)*

References

[Cooprider 79]
Cooprider.
The Representation of Families of Software Systems.
PhD thesis, Carnegie-Mellon University, April, 1979.

[DeRemer and Kron 76]
DeRemer and Kron.
Programming-in-the-Large Versus Programming-in-the-Small.
IEEE Transactions on Software Engineering SE-2(2):80-86, June, 1976.

[Feldman 79]
Feldman.
Make - A Program for Maintaining Computer Programs.
Software - Practice and Experience 9(3):pp. 255-265, March, 1979.

[Johnson 78]
Johnson.
Lint, a C Program Checker.
Technical Report, Bell Laboratories, July, 1978.

[Mitchell 79]
Mitchell, Maybury, Sweet.
Mesa Language Manual.
5.0 edition, XEROX PARC, 1979.

[Schmidt 82]
Schmidt.
Controlling Large Software Development In a Distributed Environment.
PhD thesis, University of California Berkeley, December, 1982.
This thesis is available as XEROX PARC Technical Report CSL-82-7

[Thomas 76]
Thomas.
Module Interconnection in Programming Systems Supporting Abstraction.
PhD thesis, Brown University, 1976.

[Tichy 80]
Tichy.
Software Development Control Based on System Structure Description.
PhD thesis, Carnegie-Mellon University, January, 1980.

[Weinreb and Moon 81]
Lisp Machine Manual.
4 edition, Massachusetts Institute Technology , 1981.
Chapter 24 Maintaining Large Systems