View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by AIS Electronic Library (AISeL)

Association for Information Systems
AIS Electronic Library (AISeL)

ICIS 1991 Proceedings

International Conference on Information Systems (ICIS)

1991

FORMAL SPECIFICATIONS AND COMMAND MODELING IN SOFTWARE SYSTEMS WITH A COMPLEX COMMAND STRUCTURE

Michael V. Mannino
University of Washington

In Jun Choi
Pohang Institute of Science and Technology

Sukumar Rathnam
The University of Texas at Austin

Follow this and additional works at: http://aisel.aisnet.org/icis1991

Recommended Citation

Mannino, Michael V.; Choi, In Jun; and Rathnam, Sukumar, "FORMAL SPECIFICATIONS AND COMMAND MODELING IN SOFTWARE SYSTEMS WITH A COMPLEX COMMAND STRUCTURE" (1991). *ICIS 1991 Proceedings*. 31.
http://aisel.aisnet.org/icis1991/31

This material is brought to you by the International Conference on Information Systems (ICIS) at AIS Electronic Library (AISeL). It has been accepted for inclusion in ICIS 1991 Proceedings by an authorized administrator of AIS Electronic Library (AISeL). For more information, please contact elibrary@aisnet.org.

FORMAL SPECIFICATIONS AND COMMAND MODELING IN SOFTWARE SYSTEMS WITH A COMPLEX COMMAND STRUCTURE

Michael V. Mannino
Department of Management Science
University of Washington

In Jun Choi
Department of Industrial Engineering
Pohang Institute of Science and Technology

Sukumar Rathnam
Department of Management Science and Information Systems
The University of Texas at Austin

ABSTRACT

Commands are an important part of large scale industrial software specifications, especially where the specification is separated from its implementation as in open software standards. Commands can be complex because of large numbers of parameters, dependencies among parameters, subtle side effects, and lack of abstraction. We present a formal approach for command modeling and apply it to IBM's Distributed Data Management Architecture (DDM), a complex, large scale specification of data access on remote and heterogeneous IBM systems.

Our approach consists of three parts: a declarative, executable command specification language, an incremental specification technique, and automated reasoning tools. The command specification language provides a formal interpretation of the structural (input-output) and behavioral properties (state constraints/change) of commands. To manage the details of complex commands with numerous inter-dependent arguments, a novel incremental specification technique and several tools for incremental definition and browsing are presented. Two forms of automated reasoning are also demonstrated: type checking to ensure "well-typed" expressions and target system tracing to simulate command execution. Lessons learned from our experience with the DDM are also discussed.

"It is our position that a solid product consists of a triple: a program, a functional specification, and a proof," Edsger Dijkstra, Austin TX, 09/04/89

1. INTRODUCTION

Formal methods are assuming an increasing role in software specifications especially where there must be a clear boundary between specification and implementation. In general terms, formal methods are mathematically based techniques for describing properties of software systems as well as supporting the verification and testing of software (Wing 1990). They help in revealing ambiguities, exposing errors, and communicating complex ideas as well as in proving the correctness of system properties. In many uses of formal methods, there is a strict separation between specification and implementation. Such situations include independent quality assurance, software contracting, and open systems standards. Open systems standards, characterized by one specification and many independent implementations, is increasingly common as the computing market becomes multi-vendor and highly competitive. Formal methods are increasingly being used in major software projects such as IBM's CICS (Nix and Collins 1988), the CASE project of Praxis Systems (Hall 1990), and the Portable Common Tools Environment (Middleburg 1989).

IBM's Distributed Data Management Architecture (DDM) (Demers 1988) is an internal open systems standard for data access on remote and heterogeneous IBM systems. The DDM is a low level component of the Systems Application Architecture, a large and complex standard that spans protocols for communications,user interfaces, programming languages, and data access. Functionally, the DDM is a layered communication

architecture that controls transparent interactions between source and target systems. An application program that is DDM compliant can access remote data with access, security, concurrency, and other services being provided transparently. Structurally, the DDM is a layered collection of resources that exist in servers (i.e., subsystems).[1] Important resources include files, directories, agents, and data dictionaries. Some servers support all kinds of resources while others support only a subset of resources. Independent groups within IBM choose the subset of the DDM to implement on particular servers.

Commands are an important and complex part of large scale systems such as the DDM but are often poorly specified. The importance of commands is due to their use as external interfaces to an information system. All three groups — users, implementors and architects — must share a common understanding of commands. Commands are complex because they can have many interdependent parameters and produce subtle side effects. Despite their importance and complexity, commands are often specified imprecisely. Informal text is typically the only description of a command's behavior and dependencies among parameters. The text can be long and complex without any abstractions to manage the details. (An examination of the text found in another large scale system — the help system in UNIX — also reveals all of these problems.) Since specifications are typically informal, there are no forms of automated reasoning to analyze the consistency and completeness of specifications.

We describe the general purpose command specification tools we have developed and applied to the DDM. Our approach consists of three parts: a declarative, executable command specification language, an incremental specification mechanism, and automated reasoning tools. The command specification language (CSL) describes the structural (input and output) and behavioral (state constraints and changes) aspects of commands. The CSL is based on strongly-typed, object-oriented languages whose semantics are axiomatically defined. Closely associated with the CSL is the command lattice, a novel abstraction technique that supports incremental specification of commands. Command lattices are an extension of code sharing techniques used in object-oriented programming (Choi, Mannino and Tseng 1991).

Software tools based on automated reasoning contribute to increased quality, productivity, and understandability of a specification. The goal of automated reasoning is to derive inferences about a specification. These inferences often involve long and subtle chains of reasoning. Reasoning methods can be deductive (from facts and axioms to conclusions), inductive (from examples to conclusions), and simulative (imitative representation of one function by another). Automated reasoning transforms the role of specification tools from that of merely representation to that of aiding problem solving. We focus on two automated reasoning tools we have designed and developed for the CSL: the type checker (TC) that ensures expressions are "well-typed" and the target system tracer (TST) that simulates a specification by checking whether preconditions are satisfied and subsequently asserting postconditions.

Section 2 reviews related research on formal methods in software development. Section 3 discusses the nature of commands in large scale specifications using the DDM as a representative example. Section 4 presents our tools for command modeling including the CSL, incremental definition and browsing tools, and the TC and the TST. Examples from the DDM are used to depict the use of our approach to a large and complex specification. Section 5 discusses our experience with applying our tools to the DDM and future research directions.

2. RELATED WORK

In this section, we review research on formal methods in software development. We begin with a discussion of formal specification languages and associated software tools. We then summarize research on type inference and polymorphism which has influenced our development of the TC and command lattice.

In selecting a technique for modeling commands, one should seek the following properties (Jacob 1983):

1. The specification of the command should be easy to understand. In particular, it must be easier to understand and take less effort to produce that the software that implements it.

2. The specification should be precise. It should leave no doubt as to the behavior of the system for each possible combination of parameters.

3. It should be easy to check for consistency.

4. The technique should be powerful enough to express non-trivial system behavior with a minimum of complexity.

5. It should separate what the system does (function) from how it does it (implementation). The technique should make it possible to describe the behavior of the command, without constraining the way in which it will be implemented.

6. The specification should directly yield a reasonable table of contents for a user manual, but not necessarily the material in the manual.

Floyd, Hoare and Dijkstra made the initial breakthrough in formalizing specifications declaratively by developing the concepts of Hoare triples and axiomatic semantics (Dijkstra 1976). In the Floyd-Hoare-Dijkstra model (which we adapt for our study), the behavioral properties of commands can be formally described in terms of a Hoare triplet <P {S} Q>. This states that if the pre-condition P holds in the current state of the system then the post-condition Q is guaranteed to hold provided the command {S} successfully terminates.[2] Hence, pre- and post-conditions can be described as well formed formulae and commands as predicate transformers on pre- and post-conditions. Jacob (1983) describes the chief value of this formal specification methodology as "[permitting] a designer of large and complex systems to describe the external behavior of the system without having to specify its internal implementation."

In practical terms, post-conditions declaratively describe general assertions about the values that the relevant variables will take after execution of the command. These assertions will not ascribe particular values to each variable, but will rather specify certain general properties of the values and the relationships holding between them (Hoare 1969). Hence, formalizing command specifications emphasizes the expression and manipulation of the results of computational processes and the objects on which they are performed, rather than instructions for carrying them out (Winograd 1979). This allows an automated reasoning tool to infer properties about these command descriptions (Hoare et al. 1987).

In practice, the use of a formal method (like axiomatic semantics or Hoare logic) requires the use a specification language and a collection of software tools based on automated reasoning to define and analyze specifications. Wing (1990) classifies specification languages as model-oriented versus property-oriented. Model-oriented languages feature a small set of mathematical structures such as tuples, relations, and sets to directly describe a system's behavior. Property-oriented languages feature axioms to indirectly describe properties that a system must satisfy. Axioms can be used to describe both state dependent and state independent properties of a system. VDM (Jones 1986) and Z (Spivey 1988) are prominent model-oriented languages, while Larch (Guttag, Horning and Wing 1985) is an important property-oriented language. The CSL has characteristics of both model-oriented and property-oriented languages.

Associated software tools include the usual syntactic analysis as well as simulation and proof checking tools. Simulation tools make a specification executable. Users can get immediate feedback about the meaning of a specification by executing and tracing its effects. Proof checking tools ensure that certain axioms hold in a specification and provide useful inferential information. Statemate (Harel 1988) is an example of a simulation tool for state transition diagrams, while the Larch prover (Garland and Guttag 1989) and the verifier for the Adaptable Database Programming Language (Sheard and Stemple 1989) are examples of proof checking tools. The TC is a proof checking tool and the TST a simulation tool.

Our work on command specification has also been heavily influenced by research on type inference and polymorphism in object-oriented systems (Cardelli and Wegner 1985). A type inference system ensures that expressions are "well-typed" before they are executed. "Well-typed" means that the sets of values (or types) used in expressions are guaranteed to be compatible. In other words, a type inference system will not permit illegal operations such as the addition of an integer and a string. Further, a type inference system can deduce the type of an expression, thus alleviating the user from specifying the type information. The use of type variables and subset conditions among type variables makes the type inference system flexible because only the most general type is determined at compile-time. The actual type at execution is guaranteed to be a substitution instance of the most general type. We have applied the results on type inference, described in Choi, Mannino and Tseng (1990), in the design of the CSL and the TC.

Polymorphism permits an operator to accept different kinds of arguments and respond based on the kinds of arguments received. Commands are polymorphic as they can accept different combinations of parameters. Objected-oriented programming is characterized by universal polymorphism where there is one implementation of an operator for all objects and ad hoc polymorphism where there are multiple implementations of an operator (Cardelli and Wegner 1985). Command lattices extend a form of ad hoc polymorphism known as incremental overloading (Choi, Mannino and Tseng 1991) that permits related implementations of an operator to be shared. However, in command specifications, behavioral descriptions rather than implementations can be shared.

3. COMMAND SPECIFICATION IN THE DDM

In this section, we provide background on the DDM. We begin with a general discussion of the nature of the DDM and then proceed to details of command specification.

We finish with a brief discussion of the limitations of the current DDM command specification.

The DDM has two specifications, an external one for users of IBM languages and an internal one for architects and implementors of the DDM. Externally (to users of the DDM), the DDM is a collection of commands that can be called from languages such as PASCAL, C, and COBOL. For example, a user may issue commands to declare a file as a remote file, open the file, insert records into the file, and close the file. Internally, these commands are interpreted by a DDM source translator and sent in an internal form to a DDM target translator on a remote machine. A command can cause state change on a target machine as well as send results to the source machine. The internal specification of the DDM relates to the design of the semantics (in terms of structural and behavioral properties) for DDM commands. Architects provide the structural characteristics and behavioral properties (specification) of the DDM commands, while implementors interpret the specification and code it on particular servers. Architects then test a given implementation for compliance with the specification by executing a series of test cases. Here, we are concerned with the internal description because it is complex and vital to implementors. In addition, the external description can be derived from it.

The DDM is specified in several upwardly compatible levels, comprising more than 1000 pages each. Each level defines a large collection of classes and commands (IBM 1989). A class is defined by a collection of variables. For example, the class FILE includes variables for the file name, file size, etc. To manage the large number of classes, more specialized classes inherit or share the variables of more general classes. For example, the more specialized class DIRFIL (direct file) inherits from the more general class FILE.

A command accepts arguments, causes state changes, and returns output. A command specification contains both behavioral and structural descriptions. The behavioral description expresses constraints on the state of the command's target system as well as changes to the state of the command's target system. The behavioral description of commands is typically given by bubble sheet text as shown in Figure 1 for the command CRTDIRF (IBM 1989). This text (as can be seen from Figure 1) often describes some complex conditions such as the relationship with the command DCLFIL (declare file) and the action to take when the DUPFILOP (duplicate file option) parameter is used.

The structural description of a command expresses what kind of argument values a command can take and what kind of results it returns. Figure 2 contains a partial structural specification of the DDM command CRTDIRF (IBM 1989). Note that the arguments of the command are specified as instance variables (insvar), while the output is specified as command replies (cmdrpy). Each argument has a name, type, and usage status. For example, the first argument is named crtnam, its type is an enumerated class (ENUCLS) of either a declare name (DCLNAM) or file name (FILNAM), and its status is required. In contrast, the type of the third parameter (dupfilop) is primitive, and its status is optional. The command replies define possible error messages when a problem occurs in processing a command. The reply CMDCMPRM is returned when the command completes successfully.

There are four limitations of current DDM command specifications, as depicted in the examples just presented.

1. The structural characteristics and behavioral properties of commands are informally (and therefore imprecisely) specified in detailed text. For example, the DDM command OPEN has more than four pages of help text explaining such complex constructs as side effects related to open file lists, cursors, shadow files, and lock tables.

2. There are no abstraction techniques such as inheritance to aid the understanding of the structural characteristics and behavioral properties of commands. Commands must be specified and browsed in entirety. Abstraction techniques should permit incremental specification and browsing, beginning with the most essential details and gradually adding nonessential details as desired.

3. The DDM specification is weakly typed and does not support inference of type information. Strong typing and type inference are foundations of our approach to formal specification because they promote succinct, precise, and flexible specifications.

4. The DDM lacks a declarative, executable specification. A declarative specification for the structural and behavioral properties of commands expresses the relationships between the inputs (i.e., initial state), internal variables, and outputs (i.e., resulting state) of the command in a process independent manner (Winograd 1979). Execution of a specification relates the state of a target system to the details of command specification.

These limitations are addressed by our modeling and software tools, described in the following section.

```
CRTDIRF      create direct file
This command creates a direct file on the target system. If the target agent has received a DCLFIL command
for the name specified by DCLNAM (equals DCLNAM on DCLFIL), the FILNAM parameter specified on
the DCLFIL is used by the target agent as the name for the created file.
...
LOCKING:
...
When the recreate option of DUPFILOP is executed, any previously acquired locks on the file are lost.
...
EXCEPTIONS:
If a file exists on the target system with the same name, the DUPFILOP parameter specifies the action to take
for this condition. If the DUPFILOP specified requires locks to be obtained on the file and the file is locked
by another user, FILIUSRM is returned.
...
```

Figure 1. Partial Description of the Behavioral Properties of CRTDIRF

```
insvar                                                    INSTANCE VARIABLES
crtnam    ENUCLS   DCLNAM
          ENUCLS   FILNAM
          REQUIRED
reclen    CLASS    RECLEN — record length
          REQUIRED
          NOTE     For both fixed and varying length records,
                   this is the maximum record length.
dupfilop  CLASS    DUPFILOP — duplicate file option
          OPTIONAL
...
cmdrpy                                                    COMMAND REPLIES
agnprmrm  CLASS    AGNPRMRM — permanent agent error
cmdcmprm  CLASS    CMDCMPRM — command processing completed
          NOTE     Required if no other reply object or reply
                   message is returned.
...
```

Figure 2. Partial Description of the Structural Properties of CRTDIRF

4. COMMAND MODELING TOOLS

In this section, we demonstrate salient aspects of our approach to command modeling using representative examples from the DDM.[3] For the CSL, we describe type expressions to model the structural aspects of commands and pre/post-conditions to model the behavioral aspects. After discussing the CSL, we describe the principles of the command lattice and the incremental definition and browsing tools based on the command lattice. Finally, we present the TC and TST, two automated reasoning tools for the CSL.

4.1 The Command Specification Language

Sentences in the CSL are divided into two categories: type expressions for the structural aspects of commands and pre/post-conditions for the behavioral aspects. Type expressions are formed from basic types, structured types,

type variables, and Boolean conditions on type variables. The structured types are comprised of labeled records denoted by "*" to represent classes, labeled unions denoted by "+" to represent mutually exclusive types, function types denoted by "->" to represent commands, and lists denoted by "[]" to represent collections. Type variables and Boolean conditions on type variables specify relationships among required and optional command parameters. DDM commands are polymorphic because they can accept different combinations of parameters.

Figure 3 displays the CSL type expression of CRTDIRF. The type expression in Figure 3 indicates that CRTDIRF is a polymorphic function with type variables t1 and t2. The subtype condition (≤)[4] on type variable t1 indicates that the command input must contain the arguments crtnam and reclen. Note that the type of crtnam is a labeled union indicating that either a declare name (dclnam) or file name (filnam) can be used. The supertype condition on t1 (≥) indicates that the input can optionally contain other arguments such as filinisz (file initial size) and reclencl (record length class). Note that identifiers in capital letters denote type constants, e.g, INT for the integer type and DATE for the labeled record type month::INT * day::INT * year::INT. The output of CRTDIRF is a labeled union with label cmdcmprm indicating a successful execution or label errorcmdrpy indicating an unsuccessful execution. The type of an error reply is represented as a labeled record type that must be supertype of labeled record type (agnprmrm::AGNPRMRM *...* dupfilrm::DUPFILRM *...) since it can contain several error codes with each error code appearing only once.

In the specification of some commands, there are more dependencies among parameters than merely whether a parameter is required or optional. For example, in the DDM command OPEN, the parameter filshr (file sharing) must be specified if prpshd (prepare shadow) is specified. In INSRECEF (insert record at end of file), the parameters keyvalfb and recnbrfb must be specified together. These dependencies can be indicated with conditions on type variables as shown in Figure 4.

The CSL can also be used to model classes as labeled record types with a label for each instance variable of a class. Figure 5 lists some of the most important DDM classes including AGENT, SERVER, ACCMTH (access manager), FM (file manager), and CURSOR. An agent acts on behalf of a user to submit commands to a target server. Agents contain the declare name list (dclnamls) and open file list (opnfills) of their users. A server contains the list of supported managers such as access managers and file managers. Access managers contain cursors for each user of a file where a cursor contains the position of the current record, an access intent list, and locks. File managers maintain attributes of files such as names, end of file numbers, and file expiration dates as well as the actual content of files in object lists (objlst).

Behavioral aspects of commands are modeled by pre- and post-conditions. A condition is a named Boolean formula with identifiers, comparison predicates, and simple functions to manipulate lists, labeled records, and labeled unions. For example, list functions include construction denoted by ":", tail denoted by "TL", and head denoted by "HD". Comparison predicates include equality denoted by "=", and inequality denoted by "≠". Identifiers are either local or global. The main difference between these type of identifiers is their scope. Changes to global identifiers, denoted by NEW, persist to the execution of other commands. Changes to local identifiers, denoted by ":=" do not persist after command execution.

Figure 6 lists some of the pre- and post-conditions of the CRTDIRF command. The global and local identifier definitions precede the pre- and post-conditions. There are two global identifiers: "a" denoting an instance of AGENT and "s" denoting an instance of SERVER. Condition names are separated from their associated expressions by a colon (:). The pre-condition FILE-DECLARED-OR-NOTCREATED is rather complex. It states that if a declared name is given, the file must have been previously declared: the declared name list (dclnamls) of the agent must contain the crtnam parameter. Otherwise, if the file name is given, it must not exist: there must not exist a member of the server's

```
crtdirf :: t1 -> (cmdcmprm::CMDCMPRM + errorcmdrpy::t2)
RESTRICTED_BY t1 ≤ (crtnam ::(dclnam :: STR + filnam :: str)* reclen :: INT)
AND t1      ≥ (crtnam :: STR * reclen :: INT * filinisz :: INT * reclencl :: STR
              * recdelcp :: BOOL * dupfilop :: STR * ... )
AND t2      ≥ (agnprmrm :: AGNPRMRM * mdathrm :: CMDATHRM * cmdnsprm :: CMDNSPRM
              * dclnfnrm :: DCLNFNRM * dupfilrm :: DUPFILRM * ... )
```

Figure 3. CSL Type Expression for CRTDIRF

```
open :: t1 -> (cmdcmprm::CMDCMPRM + errorcmdrpy :: t2)
RESTRISTED_BY t1 ≤ (dclnam::STR * accmthcl::ACCMTHCL * accintls::ACCINTLS)
AND t1 ≥ (dclnam::STR * accmthcl::ACCMTHCL * accintls::ACCINTLS *
        filattrl::INT * filshr::INT * prpshd::BOOL)
AND     IF t1 ≤ (dclnam::STR * accmthcl::ACCMTHCL *
            accintls::ACCINTLS * prpshd::BOOL)
        THEN t1 ≤ (dclnam::STR * accmthcl::ACCMTHCL *
            accintls::ACCINTLS * filshr::INT * prpshd::BOOL)
AND...
```

Figure 4. CSL Type Expression for OPEN

```
AGENT ::      (title :: STR * help :: STR * dclnamls :: [DCLFIL] * opnfills
              ::[STR] * path :: STR)
DCLFIL ::     (filnam :: STR * dclnam :: STR * filexnsz :: INT *
              filmaxex :: STR * agnnam :: STR)
SERVER ::     (title :: STR * help :: STR * agents :: [STR] * accessManagers
              :: [ACCMTH] * fileManagers :: [FM] *... )
CURSOR ::     (position :: INT * accintls :: [STR] * mltupders :: INT *
              locks :: [STR])
FM ::         (filnam :: STR * fileunion :: (seqfil :: SEQFIL +
              dirfil :: DIRFIL +... ))
DIRFIL ::     (title :: STR * help :: STR * eofnbr :: INT * filexpdt :: DATE
              * filinisz :: INT * filsiz :: INT * reclen :: INT *
              objlst :: [STR] ...)
```

Figure 5. Type Expressions for Some DDM Classes

fileManagers list with a file name matching the crtnam parameter. In the pre-condition FILE-DECLARED-OR-NOTCREATED, the function PROJECT denoted by "." returns the result of the extraction of a value from a component of a labeled record, the function IN tests whether a labeled union contains a particular label, and the function IS extracts a value from a labeled union.

The first four post-conditions (RECLEN-UPDATE, EOFNBR-UPDATE, FILEXNSZ-UPDATE, and MAXARNB-UPDATE) assert values for components of a DIRFIL object.[5] The LRCREATE function builds a labeled record with a label reclen and a value equal to that of the command parameter reclen. The LRAPPEND function extends a labeled record with another label/value pair. In conditions EOFNBR-UPDATE through MAXARNB-UPDATE, the result of LRAPPEND is assigned to the local identifier po1. The next two post-conditions (FILNAM-UPDATE and FILEUNION-UPDATE) assert new values for an FM object. The function IF returns a new labelled record whose initial depends on whether the command was specified with the parameter dclnam or the parameter filnam. The AS function constructs a labeled union with value po1 representing a DIRFIL and label dirfil. The final post-condition (NEW-DIRFIL) asserts a new value for the fileManagers component of the global identifier "s".

An important aspect of our approach is that dependencies among commands are stated declaratively through pre- and post-conditions. A dependency exists when a pre-condition can be satisfied by post-conditions of another command. For example, it is possible to express the constraint that a file can be opened only if it has been declared, or the constraint that a credit can be issued only if a cheque has cleared. In the case of the DDM, the declare file command (DCLFIL) must precede a CRTDIRF command if the CRTDIRF uses a declare name. The dependency can be inferred by comparing the post-conditions of DCLFIL with the pre-conditions of CRTDIRF.

```
global identifiers
a :: AGENT; s :: SERVER
local variables
pr1 :: DCLFIL; pr2 :: FM; po1 :: DIRFIL; ...; po14 :: FM
pre-conditions
-- conjunction of conditions
FILE-DECLARED-OR-NOTCREATED :
    ((a.dclnamls CONTAINS pr1) AND (pr1.dclnam = (IS crtnam dclnam)))
    OR (NOT ((s.fileManagers CONTAINS pr2) AND
        (pr2.filnam = (IS crtnam filnam))))
post-conditions
-- conjunction of conditions
RECLEN-UPDATE : po1 = LRCREATE (reclen := reclen)
EOFNBR-UPDATE : po1 := (LRAPPEND po1 eofnbr 1)
FILEXNSZ-UPDATE : po1 := (LRAPPEND po1 filexnsz NIL)
MAXARNB-UPDATE : po1 := (LRAPPEND po1 maxarnb 0)
... -- post-conditions to set default values for other DIRFIL variables
FILNAM-UPDATE : IF (IN crtnam dclnam)
    THEN po2 = LRCREATE (filnam := pr1.filnam)
    ELSE po2 = LRCREATE (filnam := (IS crtnam filnam)
FILEUNION-UPDATE : po2 := LRAPPEND po2 file (AS po1 dirfil))
-- Add a new FM to the server variable fileManagers.
-- The ':' is the list constructor function.
NEW-DIRFIL : s := LRREPLACE s fileManagers (po2 : s.fileManagers)
```

Figure 6. Pre- and Post-Conditions of CRTDIRF

As illustrated, the CSL provides a declarative way to precisely describe structural and behavioral aspects of commands (in our example, the DDM command CRTDIRF). The CSL specifications should convey much more information to architects and implementors than command help text alone. Pre- and post-conditions for large-scale specifications such as the DDM will be complex. This contrasts with most examples where formal specifications are rather simple such as for push and pop for stacks or insert and delete for queues. A source of complexity in commands not depicted so far in this section is the number and dependencies among parameters. Subsection 4.2 presents some modeling and software tools to manage this complexity. An important advantage of the CSL is that it is amendable to automated reasoning. Subsection 4.3 presents two automated reasoning tools for the CSL.

4.2 Incremental Definition and Browsing Tools

We now describe two tools that help to manage the complexity of command specifications. We discuss the nature of the complexity of commands and the relationship of command lattices to incremental overloading in object-oriented programming. We then present the incremental definition mechanism and browsing tools building on the example from the previous sub-section.

The problem addressed here is the complexity of command usage. This complexity is due to a large number of inter-dependent parameters, each having subtle side effects. For example, CRTDIRF has a dozen parameters. Other commands such as OPEN have subtle dependencies among parameters. Similar complexity can be found in the commands of other large specifications such as the UNIX operating system. It was our conviction that complexity can be managed with proper abstraction. Our incremental specification technique initially focuses on the smallest part of a command, i.e., the behavior of the command with only the required parameters. Complexity in behavior is introduced through the addition of optional parameters. However, behavior changes from optional parameters are not always monotonic as the presence of an optional parameter can alter the default effect.

In general, a command can be viewed as a collection of behaviors, one for each valid combination of parameters, controlled by a case statement to select the appropriate

behavior based on the actual combination of parameter values. In the CSL, a behavior corresponds to a collection of pre- and post-conditions for a combination of parameters. An obvious problem with this interpretation is the number of individual behaviors that must be defined. In the worst case, the number of behaviors is the size of the powerset (set of all subsets) of a command's parameters. However, the worst case or anything near the worst case is unacceptable because the command specifications would be too long to write and understand. To be manageable, a command must be completely described by combinations of a small set of behaviors from which all possible cases can be derived. The burden of command specification can then be reduced because behaviors can often be shared among the set of minimal behaviors.

We formalize these notions by introducing the concepts of a command lattice and a minimal command tree.[6] A command lattice is a set of parameter subsets partially ordered by set inclusion with a least upper bound and greatest lower bound.[7] Each node of a command lattice is a parameter subset associated with a collection of pre/post-conditions that describes its behavior. The root of the lattice is the required parameters, and the terminal node is the set of all parameters.

A minimal command tree is a sub-graph of the command lattice that contains the same root node and satisfies two properties: disjoint, and complete. It is defined as follows:

Complete: $\bigcup_{1 \ldots N} (\text{unique}(i)) = \text{TERM}$

Disjoint: $\bigcap (\text{unique}(i), \text{unique}(j))\ \phi$
$\forall\ i,j \text{ in TREE}, i \neq j$

where
- TERM is the terminal node in the command lattice,
- TREE is the set of N nodes in the minimal command tree,
- unique(i) is the unique parameters in node i = (PARAM(i) − PARAM(j)), j = PARENT(i),
- PARAM(i) is the set of parameters of node i, and
- PARENT(i) is the parent of node i.

The behavior of a node in the minimal tree is defined as the union of the behaviors of the node's parent, the new pre/post-conditions of the node, and the overridden pre/post-conditions of the parent. Since the behavior of the parent node can be inherited, only the new and overridden behavior need be specified. It must be noted that the command lattice is a lattice of the structural properties of commands and that the behavioral properties of commands do not constitute a lattice.

Command lattices extend incremental overloading of object oriented programming in two aspects. First, the code sharing in incremental overloading is between a subclass and the ancestor classes on one path, while code sharing in command lattices involves multiple parents. Second, code sharing for incremental overloading is total and procedural. If there is any undesirable effect from an ancestor's code, the code in the subclass must override it. Overriding is difficult because the code is procedural and the programming environment does not usually show the shared code. In contrast, sharing in command lattices is explicit through named pre/post-conditions.

We have implemented two tools that support the definition of minimal command trees, inheritance and overriding of pre/post-conditions: syntax directed editors for pre/post-conditions and the browsers for command lattices. Figure 7 displays the command definition pane for CRTDIRF. The left subpane lists the command components that can be defined. Generally, a user proceeds by defining the minimal command tree in two steps. First, the parameter names and types are specified and divided into required parameters (root of the command tree) and optional parameters. Second, the user completes the minimal tree by defining rules among optional parameters using an AND/OR notation. The resulting minimal command tree is checked for disjointness and completeness.

Pre/post-conditions are specified for nodes of the minimal command tree. The user selects parameters from the middle panes and then pulls down a menu to define, delete, or browse pre/post-conditions for the given combination of parameters. Only valid combinations of parameters as defined by the minimal command tree can be selected. Pre/post-conditions from ancestor nodes are inherited. The user needs only to define new and overridden pre/post-conditions. Overriding is indicated by defining a condition with an identical name as an inherited condition.

Pre/post-conditions are defined with a syntax directed editor. Figure 8 displays the completed pre-condition FILE-DECLARED-OR-NOTCREATED for CRTDIRF. To avoid precedence rules and parentheses, the editor supports prefix construction of expressions. The user selects tokens from a menu of term types. In Figure 8, note NOT with one argument required and AND with two arguments required. Depending on the term type chosen, the editor may prompt for another input or display another menu of options. For example, if PROJECT is chosen and the user selects a variable, the editor prompts for the component in the labeled record

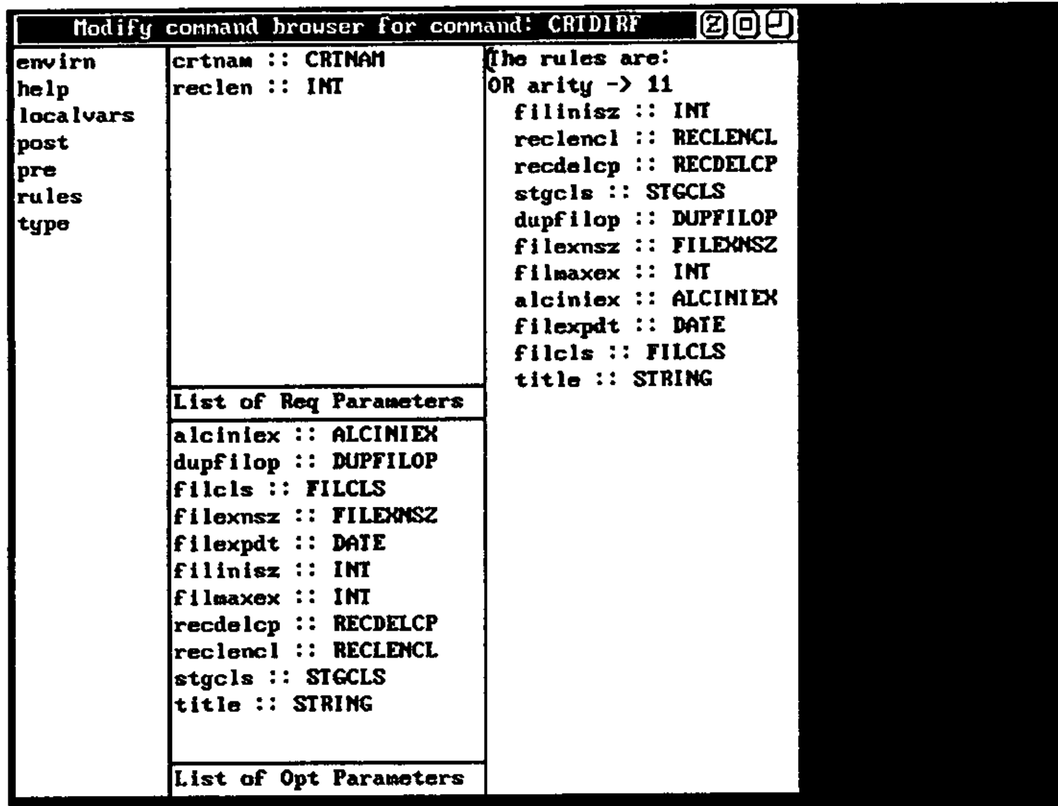

Figure 7. Modify Browser for CRTDIRF

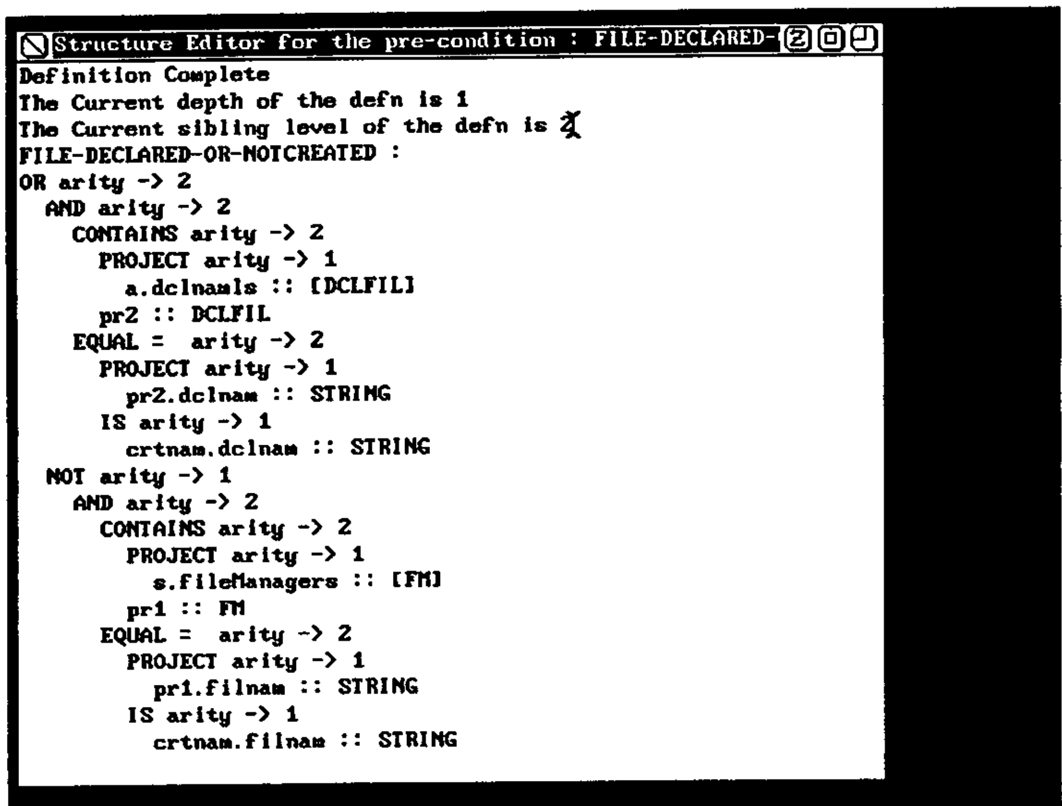

Figure 8. Structure Editor for FILE-DECLARED-OR-NONCREATED

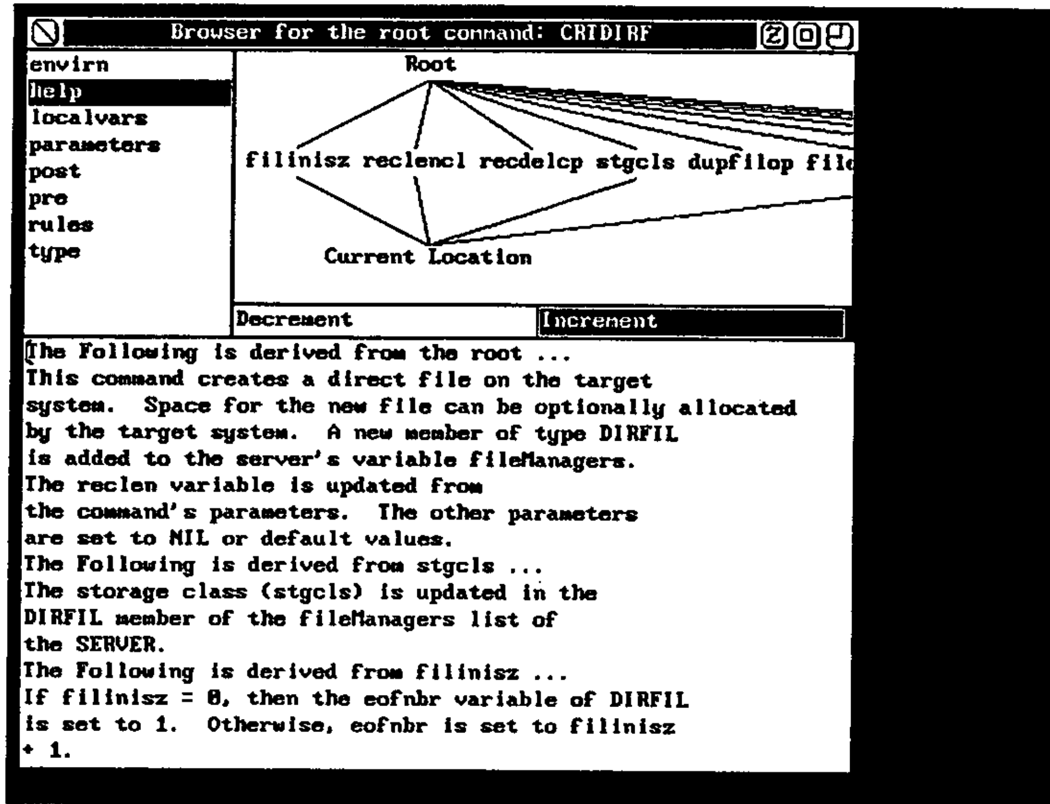

Figure 9. Browser for CRTDIRF

type of the variable. When the user selects PROJECT from the term type menu and the variable pr1 from which to project, the user is presented with a menu of the components of DCLFIL: the type of pr1.

A user may view the pre/post-conditions for any node of a command lattice using the Command Browser tool. Figure 9 displays a command browser for CRTDIRF. The right pane contains a graphic representation of the command lattice. Initially the lattice is set to the root parameters. A user may add/remove optional parameters by selecting the increment/decrement buttons. Only valid combinations of parameters may be selected. For each parameter selected, an arc is added to the graph. After selecting a combination of parameters, the corresponding collection of pre/post-conditions can be viewed by selecting a command attribute from the left pane.

4.3 Automated Reasoning Tools

We have developed two tools, the type checker (TC) and the target system tracer (TST) to demonstrate the value of automated reasoning for formal specification languages. The theoretical basis for the design of the TC and the TST are derived from denotational semantics (Scott and Strachey 1971) and λ-calculus (Church 1941), axiomatic semantics (Hoare 1969), and strongly typed object oriented languages (Mannino, Choi and Batory 1990).

4.3.1 The Type Checker

The TC is a deductive reasoning tool using facts and axioms about set membership and inclusion (subset). Recall that types are sets constructed from basic sets such as integers and structured sets such as the set of all lists of integers. Facts about set membership and inclusion are represented by two constraint sets:

- The assignment set containing associations of an identifier with a type expression denoting that bindings of the identifier are members of the set denoted by the type expression,
- The restriction set containing inclusion relationships among type expressions.

The latter set contains the constraints that define dependencies among the parameters of a command. Given a

collection of type constraints, a type inference system can answer two kinds of questions using rules of inference. First, a type inference system can determine whether an expression is "well-typed." Second, a type inference system can deduce unknown type information such as identifiers with unspecified types and sub-expressions. Furthermore, type inference systems are usually designed to deduce the principal or most general type.

The TC can be used in two contexts for expressions in the CSL. First, when a user type checks an instantiated command, the type inference system tells whether the command is well-typed. For example, the type inference system gives the replies shown in Figure 10 for instantiations of CRTDIRF.

Since the first example in Figure 10 is well-typed, the TC responds with the known range type of CRTDIRF. The type identifiers in the type expression can be expanded on demand. The second example is not well-typed because the reclen parameter is required. The TC ascertained that the absence of this parameter did not satisfy the inclusion constraints given for CRTDIRF (Figure 3).

The second context for the TC is for pre/post-conditions. Here, a user wants to check whether a condition is well-typed and the types of the identifiers used in the condition. For the pre-condition, FILE-DECLARED-OR-NOTCREATED (repeated below in Figure 11), the TC infers the type of the entire condition and the types of the identifiers, pr1 and pr2. It is not necessary for the user to explicitly define the types of identifiers.

The details of the TC are beyond the scope of this paper. For a detailed treatment of type inference including the rules of inference, complexity analysis, and formal semantics consult Choi, Mannino and Tseng (1990).

4.3.2 The Target System Simulator

The TST complements the TC. The TC ensures that expressions have meaningful values, but it does not make inferences about pre/post-conditions. The TST simulates the effects of pre/post-conditions and describes the state of the target system. Because commands cannot be executed, the TST operationalizes the execution of commands by checking whether pre-conditions are satisfied and subsequently asserting post-conditions. The ability to simulate pre/post-conditions can benefit both implementors and architects of any large scale specification. Implementors can compare the execution trace of their system against the abstract target system. Architects can increase their understanding of pre/post-conditions and discover subtle dependencies among commands.

Figure 12 illustrates tracing of CRTDIRF and OPEN command instances. The CRTDIRF command succeeds as shown in Figure 12. The pre/post-conditions are listed first followed by a trace of state changes in the abstract target system. In this case, the value of s.fileManagers has changed where "s" represents an instance of SERVER. The old value was an empty list and the new value is a list with a single labeled union value denoted by [AS dirfil ...]. The OPEN command fails because the pre-condition FILE-DECLARED was not satisfied.

To investigate the failure of the OPEN command, one can display the pre-condition and examine the state of the target system as shown in Figure 13. FILE-DECLARED failed because the dclnamls component of the AGENT instance "a" is empty.

Internally, the TST relies on the inference engine of the Smalltalk/V286 (Digitalk 1988) Prolog class. Pre/post-conditions specified by an architect are automatically

```
crtdirf(crtnam := 'file1' * reclen := 100 * filexpdt := (month := 2 *
    day := 20 * year := 91)) :: (cmdcmprm::CMDCMPRM + errorcmdrpy::t2)
    RESTR_BY t2 ≥ (agnprmrm :: AGNPRMRM * cmdathrm :: CMDATHRM *
    cmdnsprm :: CMDNSPRM * dclnfnrm :: DCLNFNRM * dupfilrm :: DUPFILRM *
    ...)
    is well typed
crtdirf(crtnam := 'file1' *
    filexpdt := (month := 2 * day := 20 * year := 91))
    is ill-typed.
```

Figure 10. Type Inference for Instantiations of CRTDIRF

```
FILE-DECLARED-OR-NOTCREATED :
    ((a.dclnamls CONTAINS pr1) AND (pr1.dclnam = (IS crtnam dclnam)))
    OR (NOT ((s.fileManagers CONTAINS pr2) AND
        (pr2.filnam = (IS crtnam filnam))))

The result is:
FILE-DECLARED-OR-NOTCREATED :: BOOL
pr1 :: DCLFIL; pr2 :: FM
```

Figure 11. The Pre-Condition FILE-DECLARED-OR-NOTCREATED

```
Input to the Target System Tracer is:
crtdirf(crtnam := 'file2' * reclen := 50)

The outcome is...
Testing Pre-Conditions: FILE-DECLARED-OR-NOTCREATED
Asserting Post-Conditions: RECLEN-UPDATE; MAXARNB-UPDATE; EOFNBR-UPDATE; ...;
FILEUNION-UPDATE; NEW-DIRFIL

All post-conditions asserted
Pre-conditions satisfied -- Post-conditions asserted.

State Tracing for Command: CRTDIRF
new value of s.fileManagers:
file := [ AS dirfil (filsiz:=0 * objlst:=[] * stgcls:=NIL * reclencl:=NIL * recdelcp:=NIL * filmaxex:=0 *
filexnsz:=0 * title:=NIL * eofnbr:=1
 * maxarnb:=0 * reclen:=50) * filnam:='file2')]

old value of s.fileManagers: []

Input to the TST is :
open(dclnam := 'alias2' * accintls := 'R' * accmthcl := 'DIR')

The outcome is...
Testing Pre-Conditions: FILE-NOTOPENED; FILE-DECLARED

Pre-condition FILE-DECLARED failed.
```

Figure 12. Command Tracing for CRTDIRF and OPEN

```
FILE-DECLARED : (a.dclnamls CONTAINS pr1) AND (pr1.dclnam = dclnam)
The agent is: title:=NIL * dclnamls:=[] * opnfills:=[] * path:=NIL
```

Figure 13. Trace Back of Cause of Failure for OPEN

translated into Prolog clauses and Prolog's inference engine checks pre-conditions, binds variables, and asserts post-conditions. A detailed description of the augmented Smalltalk/V286 Prolog structures and predicate definitions that correspond to the basic data types and operators of the CSL, translation and code generation algorithms is given in Mannino, Choi and Rathnam (1991).

5. EXPERIENCES, CONCLUSIONS, AND DIRECTIONS FOR FUTURE RESEARCH

The software tools described in Section 4 have been implemented in a mixture of Smalltalk/V286 and Smalltalk/V286 Prolog. The TC was implemented in Smalltalk/V286 Prolog; the other tools including the TST code generator, syntax directed editors, command browser, and direct manipulation graphical user interface were implemented in Smalltalk/V286. Extensive context sensitive help is provided for most functions in the prototype. The prototype is in its third version and the Smalltalk/V286 code is approximately 15,000 lines, 250,000 bytes, 35 classes, and 700 methods.

Our experience with the prototype, though limited to a subset of DDM commands, has been positive. We have defined six commands comprising more than 70 conditions. Many of these conditions, such as those for CRTDIRF, were complex and required careful analysis. In the process of modeling the commands, we discovered that a specification language has to be very expressive to be useful. Limiting the expressive power by removing disjunction, negation, or selected functions makes reasoning more efficient but renders many interesting conditions practically impossible to express.

Since our focus is on using the CSL to specify a large and complex information system, we extended the syntax and expressive power of the CSL in a number of ways. For example, we incrementally extended the CSL with some (non-primitive) functions including LRAPPEND (labeled record append), LRREPLACE (labeled record replace) and IF-THEN-ELSE. We are now convinced, as eloquently argued by Doyle and Patil (1989), that "expressive power is the key for large-scale specifications." Efficient forms of automated reasoning should be developed to fit the desired level of expressive power rather than restricting the expressive power to available forms of automated reasoning.

Our experience with the prototype has also revealed the problems of transferring our technology. One problem is that IBM has been using the current specification for several years and a number of DDM implementations have been completed. It will be difficult to justify retrofitting the current five levels of specification. More likely, IBM will identify a future level of the specification and apply our approach in a pilot study. A second problem is the cost to apply our approach. There will be additional software development costs to make our software tools production quality. Architects and implementors will require significant training in formal methods. In addition, the formal methods will probably require more time, even after training is completed.It is more difficult to be precise with our approach than imprecise with a text specification. Overcoming resistance to change and additional costs poses a significant challenge, but we hope that the long term benefits will outweigh these costs.

This work is part of our long term interest in formal methods and object-oriented modeling and their application to large scale specifications such as the DDM. We are continuing to refine and extend the prototype. Some important extensions of the CSL and prototype include class constraints, proof checking tools for pre/post-conditions, test case generation, and versions. We feel that the refinement and extension of our approach can contribute to the improved productivity of architects and implementors of software systems with complex command structures.

6. ACKNOWLEDGMENTS

This research was supported in part by a grant from IBM Corporation.

7. REFERENCES

Cardelli, L., and Wegner, P. "On Understanding Types, Data Abstraction, and Polymorphism." *ACM Computing Surveys*, Volume 17, Number 4, 1985, pp. 471-522.

Choi, I.; Mannino, M.; and Tseng, V. "Type Restrictions and Method Interfaces in Object-Oriented Database Programming." *IFIP TC2 Working Conference on Database Semantics: Object Oriented Databases*, July 1990, Windermere, United Kingdom.

Choi, I.; Mannino, M.; and Tseng, V. "Graph Interpretation of Methods: A Unifying Framework for Polymorphism in Object-Oriented Programming." *OOPS Messenger*, ACM SIGPLAN, Spring 1991.

Church, A. "The Calculi of Lambda-Conversions." *Annals of Mathematical Studies*, Number 6, Princeton, New Jersey: Princeton University Press, 1941.

Demers, R. "Distributed Files for SAA." *IBM Systems Journal*, Volume 27, Number 3, 1988, pp. 348-361.

Digitalk Inc. *Smalltalk/V286 Reference Manual.* Los Angeles, California, 1988.

Dijsktra, E. W. *A Discipline Of Programming.* Englewood Cliffs, New Jersey: Prentice-Hall, 1976.

Doyle, J., and Patil, R. "Two Dogmas of Knowledge Representation." *Technical Report Number MIT/LCS/TM-387.b*, Laboratory for Computer Science, Massachusetts Institute of Technology, September 1989.

Garland, S., and Guttag, J. "An Overview of LP, The Larch Prover." In *Proceedings of the Third International Conference on Rewriting Techniques and Applications*, Chapel Hill, North Carolina, 1989, pp. 137-151.

Guttag, J.; Horning, J.; and Wing, J. "The Larch Family of Specification Languages." *IEEE Software*, Volume 2, Number 5, September 1985, pp. 24-36.

Hall, A. "Seven Myths of Formal Methods." *IEEE Software*, Volume 7, Number 5, September 1990, pp. 11-19.

Harel, D. "On Visual Formalisms." *Communications of the ACM*, Volume 31, Number 5, May 1988, pp. 514-530.

Hoare, C. A. R. "An Axiomatic Basis for Computer Proramming." *Communications of the ACM*, Volume 12, Number 10, October 1969, pp. 576-580.

Hoare, C. A. R.; Hayes, I. J.; Jifeng, He; Morgan, C. C.; Roscoe, A. W.; Sanders, J. W.; Sorensen, I. H.; Spivey, J. M.; and Sufrin, B. A. "Laws of Programming." *Communications of the ACM*, Volume 30, Number 8, August 1987, pp. 672-686.

IBM Corporation. *Distributed Data Management Level 2.0 Architecture Reference.* January 1989.

Jacob, R. J. K. "Using Formal Specifications in the Design of a Human-Computer Interface." *Communications of the ACM*, Volume 26, Number 4, April 1983, pp. 259-264.

Jones, C. B. *Systematic Software Development Using VDM.* Englewood Cliffs, New Jersey: Prentice-Hall, 1986.

Mannino, M.; Choi, I. J.; and Batory, D. "The Object-Oriented Functional Data Language." *IEEE Transactions on Software Engineering*, Volume 16, Number 11, November 1990.

Mannino, M.; Choi, I. J.; and Rathnam S. "The Design and Implementation of Automated Reasoning Tools for the Distributed Data Management Architecture." Working Paper, Department of Management Science and Information Systems, The University of Texas at Austin, February 1991.

Middleburg, C. "VVSL: A Language for Structured VDM Specifications." *Formal Aspects of Computing*, January-March 1989, pp. 115-135.

Nix, C., and Collins, B. "The Use of Software Engineering, Including the Z Notation, in the Development of CICS." *Quality Assurance*, September 1988, pp. 103-110.

Scott, D., and Strachey, S. "Towards a Mathematical Semantics for Computer Languages." *Procceedings of the Symposium on Computers and Automata*, 21, Microwave Research Institute Symposia Series, Polytechnical Institute of Brooklyn, 1971.

Sheard, T., and Stemple, D. "Automatic Verification of Database Safety." *ACM Transactions on Database Systems*, Volume 14, Number 3, September 1989, pp. 322-368.

Spivey, J. *The Z Notation: A Reference Manual.* Englewood Cliffs, New Jersey: Prentice-Hall, 1989.

Wing, J. "A Specifier's Introduction to Formal Methods." *IEEE Computer*, Volume 23, Number 9, September 1990, pp. 8-24.

Winograd, T. "Beyond Programming Languages." *Communications of the ACM*, Volume 22, Number 17, July 1979, pp. 391-401.

8. ENDNOTES

1. The meaning of the word server in the context of this paper must not be confused with the more popular meaning of the same word in the term file server.

2. Similarliy, if a system or mechanism is denoted by S and the desired post-condition by R, then we denote the corresponding weakest pre-condition by wp(S,R). If the initial state satisfies wp(S,R), the mechanism is certain to establish the eventual truth of R. Because wp(S,R) is the weakest pre-condition, we also know that if the weakest precondition does not satisfy wp(S,R), this guarantee cannot be given (Dijkstra 1976). Hence, if <P {S} Q> can be proved, then the familiar logical symbol for theoremhood $\vdash$ P {S} Q can be used (Hoare 1969).

3. For a more detailed discussion of the CSL, consult Mannino, Choi and Rathnam (1991) where a complete syntax is given.

4. Type expression A is a subtype of B. ($A \leq B$) means that A denotes a subset of B. For labeled record types, $LR' \leq LR$ if and only if LR' contains the labels of LR and each type in LR' associated iwth a label in LR is a subtype of the corresponding type in LR.

5. The operator "=" denotes equality, the operator ":=" denotes logical assignment.

6. Note that the use of the qualifier minimal refers to the fact that the entire command lattice can be generated from this command tree. Hence, the structure of the minimal command tree is not minimized in the optimization sense. The usage is analogous to the usage of the word in database normalization in which a "minimal" cover is computed.

7. The least upper bound is the set of required parameters of a command that contains the smallest set of behaviors. The greatest lower bound is the largest set containing all the possible required and optional parameters of a command. Consequently, it contains the largest set of behaviors.