A MODEL FOR SOFTWARE REUSE IN A MULTIPARADIGM ENVIRONMENT

By
BRENDAN MACHADO
Bachelor of Science
University of Bombay
Bombay, India
1985

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
DOCTOR OF PHILOSOPHY
December, 1991

Thesis
1991D
M149m

A MODEL FOR SOFTWARE REUSE IN A MULTIPARADIGM ENVIRONMENT

Thesis Approved:

[illegible]

Thesis Advisor

[illegible]

William David Miller

[illegible]

Thomas C. Collins

Dean of the Graduate College

ACKNOWLEDGMENTS

I wish to express sincere appreciation to Dr. K. George for his encouragement and advice throughout my graduate program. Dr. W. Miller spent many hours unraveling the secrets of T_EX. Thanks go to him for his generosity with time and patience. Many thanks also go to Dr. G. Hedrick and Dr. J. Friske for serving on my graduate committee. Their suggestions and support were very helpful throughout the study.

To the Computer Science Department secretaries, Anna Ventris and Janice Bryan, who were always supportive and cheerful, I extend sincere thanks. To Terry Johnson for always keeping things in perspective; thanks go to him and to Fouad, Hisham, Jo, and Saeed for their friendship. Thanks also to Dr. R. Brown for valuable advice and a most stimulating semester.

My sisters, Rowena and Wendell, provided many uplifting conversations. My aunt Henryetta kept in touch. Her humorous correspondances helped me immeasurably. Thanks are due to them.

My parents, Raphael and Grace Machado, encouraged and supported me all the way and helped me keep the end goal constantly in sight. Special thanks go to them for their continuous faith and support.

TABLE OF CONTENTS

Chapter		Page
I.	INTRODUCTION	1
	Summary	1
	Problem Statement	1
	Importance of the Topic	2
II.	RELATED WORK	7
III.	A MODEL FOR MULTIPARADIGM SOFTWARE REUSE	22
	World Interconnection Model	22
	An Example	31
	An Introduction to Modal Logic	34
	Historical Note	36
	Contemporary Work	37
	Structure and Modality	38
IV.	THE LANGUAGE AND ITS DESCRIPTION	47
	World Interconnection Language	47
	Example 1: Multiparadigm Software Reuse	51
	Example 2: Exposure of Inheritance	53
	Example 3: Multiple Versions	57
V.	SUMMARY, CONCLUSIONS, AND FUTURE WORK	61
	BIBLIOGRAPHY	63

LIST OF FIGURES

Figure Page

1. Possibility relations from stack & queue to deque 6
2. Specification of a stack ADT [WSHF81] 25
3. Possibility relations from stack & queue to deque 27
4. Specification of a queue ADT [WSHF81] 28
5. Specification of a deque ADT (first part) 29
6. Specification of a deque ADT (second part) [WSHF81] 30
7. Possibility relations between stack, queue, & deque 30
8. Hierarchical & world interconnection models 32
9. Relations between i, r, c with view + ... 33
10. **ui, uio** relations between i, r, c with view (+ - * /) 34
11. Reflexive relations at worlds 40
12. Reflexive & symmetric relations at worlds 41
13. Reflexive & transitive relations at worlds 43
14. Reflexive, symmetric & transitive relations at stack 44
15. Paraphrased excerpt from library 52
16. Implementing stack using deque 55
17. A schema for table search 59

CHAPTER I

INTRODUCTION

Summary

Software reusability and extendibility may be facilitated by imposing an appropriate structure on software systems. Object-oriented design together with the client and heir relations has attempted to address these issues. However, there exists a vast resevoir of routines already written in different languages that may be reused. This research describes a model to interconnect multiparadigm routines using the concept of possible worlds. The structure imposed on such a system facilitates reusability and extendibility.

Problem Statement

This research uses the concept of possible worlds to develop a model for multiparadigm systems. A software system is composed of uninstantiated modules, module interfaces, and propositions, called possible worlds. Specifically, the model should provide structure to a system of multiparadigm routines. Structure, allowing "used in" **ui** and "used instead of" **uio** relations based on the concept of views, facilitates reusability and extendibility. A view is a set of interfaces and propositions.

The choice of a programming language is but one step in the software development cycle. According to Zave [Zav89], complex systems have characteristics and complexities that are inherent to the process and not to the current way of doing things. In such cases the problem characteristics may be completely satisfied only by the use of more than one programming language; e.g., one may require the use of a database query language, the sophisticated array manipulation of C or Pascal, the simple list manipulation of Lisp, and the declaratives and rules of Prolog. The characteristics of these languages are classified under different models or paradigms and systems consisting of programs from different paradigms are called *multiparadigm systems*. Each paradigm may be the basis for a class of programming languages. Some of the well known paradigms are: procedural, functional, object-oriented, logic, rule-based. While a paradigm offers a focused and cohesive view, a single paradigm may not be able to describe all the aspects of a system. Thus in a multiparadigm system different paradigms offer the raw material to solve problems.

Importance of the Topic

How easy is it to decompose a problem into manageable subproblems, fit the methods and structures required to solve each subproblem to the methods and structures available in a large library of multiparadigm codes, interconnect the encoded subproblems so that they may communicate, and execute the system? Although much work has been done, common experience shows that we continually start from scratch to build a system. Among the many reasons cited in the literature [BR87,Gog86,Pri87,Mey87] are the following:

1. We are biased towards one programming language and prefer to use it.

2. Even if we are proficient in several languages, the decomposition of a problem into subproblems and the mapping of the structures and methods to a language may be difficult and is done in an ad hoc manner.

3. It is likely that for many subproblems, solutions written in several different languages already exist in software libraries or in one of our directories. However, we tend to not reuse multiparadigm software.

In large software systems, there is the potential for a large gain in productivity if software can be reused. Lewis and Oman [LO90] predict that in the future, programs and data will be interchanged and mixed at the users request. Developers will be able to use off-the-shelf code that can interoperate. Users will have the choice of either buying or building. Users will have available large libraries of multiparadigm routines and they will have to compose and coordinate these routines to form a multiparadigm system [Zav89].

Working systems are beginning to appear for the construction of multiparadigm systems. Several advocate the use of a module interconnection language to glue together programs written in different languages [MHS86b,MHS86a,Pur86,Zav89]. Other systems advocate the marriage of paradigms and introduce new languages or new features in existing languages [Hai86a,Hai86b,Sea87].

Reusability is one of the most significant factors in improving software development productivity and quality [RGP86]. Reusability can be useful during the entire software life cycle such as specification, design, and testing. One of the problems of reuse is that code may be written in a different language. The difficulty of mixed language code reusability from large software libraries is to find a software from a description of it. This semantic retrieval is a fundamental problem of AI. The model proposed in this thesis will address the issue of integrating the selected software into the system. This involves the correct binding of interfaces.

Meyer [Mey87] makes a strong argument for designing reusable code using the object-oriented design paradigm. He argues that the problem of software reuse is a technical one and cannot be solved by the design of better libraries, library retrievals, or management. Instead, reuse is limited because designing reusable software is difficult. Attempts have been made to design programming languages to support reusability. E.g., Algol-68 and Ada offer overloading, and Ada and Clu offer genericity. These techniques are useful in developing reusable code but they do not go far enough. They are not flexible because a complex hierarchy of representations that have different levels of characterization cannot be described. They offer only two levels: generic and fully instantiated modules. Neither technique allows a client to use various implementations of a data abstraction without knowing which implementation is used in each instance.

In object-oriented design, software is constructed as a structured collection of abstract data-type implementations. An abstract data-type is a class of objects characterized by the operations available on them and the abstract properties of these operations. Object-oriented design actually identifies modules with implementations of abstract data-types. Such a dual purpose structure is called a class. Classes may be structured using two different relations: client and inheritance.

Class A is a client of B if A contains a declaration of the form bb:B. A may manipulate bb only through the features defined in the specification of B.

Class C defined as a heir to class A has all the features of A, to which it may add its own.

The powerful combination of object-oriented design and the client and inheritance relations is a key element in achieving extendibility and reusability. Unfortunately, current implementations of the object-oriented paradigm compromise these key benefits.

1. Exposure of instance variables. Consider the design of a class *stack* whose operations $push, pop, top, empty$ may be implemented in an array or with pointers. Since clients of *stack* are allowed to access the instance variables of *stack*, the designer can no longer change an instance variable without the risk of adversely affecting descendant classes that access *stack* instance variables.

2. Exposure of Inheritance. Define a class *Deque* with the following methods: $push, pop, top, empty, nq, dq, front$. Define another class *Stack* with the following methods: $push, pop, top, empty$. An undesirable effect of defining *Stack* to inherit from *Deque* is that *Stack* inherits the extra operations $nq, dq, front$ from *Deque* (Figure 1). If all operations are visible to clients of *Stack* then

switching to the self contained implementation of *Stack* (which does not support them) could be an incompatible change. Similarly, *Queue* inherits the extra operations *push*, *pop*, *top* from *Deque*.

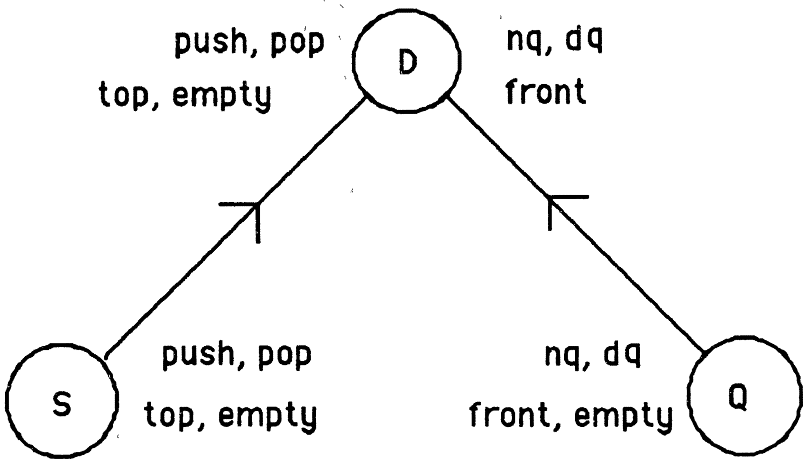

Figure 1. Possibility relations from stack & queue to deque

This research provides structure to a multiparadigm system using a possible worlds model. In the possible worlds model, a software system is composed of worlds. Each world is related to other worlds by possibility relations based on views. Both the client and inheritance relations are embodied in the possibility relation.

CHAPTER II

RELATED WORK

A panel of 15 representatives from industry and academia was asked to identify the software challenges for the next five years [LO90]. Portability across architectures, distributed transaction based computing, user programming, smart systems, interoperability, and object-oriented design are on their list. The development of heterogeneous systems has tried to address these challenges.

In a heterogeneous system, multiparadigm software communicates across a variety of architectures. Notkin [Not90] reports on the development of the Heterogeneous Computer System (HCS) at the University of Washington. The goal of HCS is to increase the sharing of services across systems, while decreasing the cost of integrating new types of systems and services into the environment. HCS provides a set of network services such as electronic mail, filing, and remote computation to a diverse collection of systems using heterogeneous remote procedure calls and naming services. Although multiple standards are accommodated economically, existing programs are not guarenteed transparent access to other existing programs. Proxy subclasses are defined to implement the different HCS services on different machines. A proxy subclass supplies a specific systems method for providing an abstract heterogeneous service. An object-oriented design is used to develop the abstract and proxy classes.

Einarsson and Gentleman [Ein85,EG84] point out that two kinds of difficulties affect mixed language programs.

1. Differences in language definition
   (a) Unique data structures; e.g., Fortran has no record structure and Pascal has no complex type.
   (b) Differing I/O abstractions or file structure abstractions; e.g., Fortran assumes discrete record I/O operations versus byte stream I/O operations for many other languages. Random access files cannot be defined in standard Pascal but are defined in many Fortran dialects.
   (c) Parameter passing semantics; e.g., In C, copy-in/copy-out semantics are implemented using the address of the operand but this is difficult for Fortran to support.
   (d) Binding time differences; e.g., Can files be opened and closed during execution or should they be attached before execution?
   (e) Exception handling. Should exceptions be carried over from the caller to the callee routine?
   (f) Asynchronism: processes, interrupts, semaphores, rendezvous .... These various models of programming with multiple processes are quite incompatible.
2. Incompatible implementation

(a) Data structure representation; e.g., Fortran stores multidimensional arrays by columns while Pascal does so by rows.

(b) I/O buffering. I/O support packages for each language tend to do their own buffering, so interaction with the users' terminal produces messages that are out of order.

(c) Labels and transfer of control. The problem is one of cleaning the environment being left and re-establishment of the execution environment.

(d) Passed procedures. A calls B and passes it C as a parameter. The problem is one of establishing an environment in which C can execute.

(e) Incompatible storage management. Details of implementation such as how the stack is organized and what pointers are kept, the levels of indirection used to access dynamically allocated blocks, etc.

(f) Separate compiled/interpreted execution. In most cases it is assumed that the basic unit of execution is a procedure. However not all language processors support separate compilation and some are interpretive.

(g) Integrated programming environments. The Lisp environment has editors and debuggers built in to support a single language. These tools may not tolerate a foreign language in the environment.

(h) Environment setup and initialization can be awkward.

Einarsson [Ein86] suggests three major methods to mix programming languages.

1. The system developer makes inter language communication possible with some of the languages supported by the manufacturer of the system.

2. A virtual program library can be written in a portable computer oriented language (PCL) and called from a user oriented language (UL).

3. Two or more programs written in different languages can interact by writing and reading ASCII files on an external memory device.

Darondeau et al. [DGR81] define communicating entities as separately compiled program components. The authors assume that some universal connections and control passing mechanisms have been defined as a common extension of every communicating language. This allows program components to be linked independent of their respective languages. The authors define two means of communication between languages.

1. Standard Types. Extending a set of languages with a common set of standardized data types provides a means of inter language communication. Each language is provided with a means of conversion between shared standard types and other local data types.

2. Foreign Types. A disadvantage of the standard type is its fine granularity. Compound types must be converted to/from standard types for inter language communication with the associated expense. For a given program component, a foreign type is one which is referenced under the generic form $M.T$ with M being the universal name of a program component which exports type T.

Foreign parameters are passed by name as in Algol-60. The only operations on a foreign type *M.T* outside *M* are those that have been explicitly exported by *M*.

Goguen [Gog86] suggests that an environment library be constructed. In it, program components from which many different but related systems may be constructed are stored. Store design information and knowledge that went into constructing the code. Stored documentation of each component in the form of formal specifications (i.e. sets of axioms) describes what each component is supposed to do. This knowledge of design objectives and decisions is stored with the code so that it is available during debugging and maintenance when 80% of the sotware effort is expended [KG87]. Program composition is achieved by the use of a library interconnection language *LIL*. A *LIL* package and *make* clauses allow for separate, independent, and incremental compilation. Each *LIL* package may have multiple versions in multiple programming languages. To construct new entities from old ones, several different approaches may be used:

1. Sew together two entities along a common interface.
2. Enrich an existing entity with some new features.
3. Hide some existing features of an entity.
4. Slice an entity to eliminate unwanted functionality.
5. Implement one abstract entity using features provided by others.

Goguen proposes the following major semantic concepts:

1. Vertical composition involving top-down and/or bottom-up hierarchy of abstract machine levels.

2. Horizontal composition involving modularization at a given level.

3. Theories which declare the properties that an actual parameter must have to meaningfully substitute for the formal parameter of an entity.

4. Views that describe semantically correct bindings at interfaces and thus describe interconnections.

Balzer [Bal71] defines Ports as a data element used for communication with files, terminals, physical devices, other programs, and the monitor. One logical implementation is with Incremental System Programming Language *ISPL*. In this implementation, Ports are defined in terms of data semaphores, which are Dijkstra semaphores with data. A Port data type consists logically of a pointer to the Port to which the connection is made and a data semaphore representing the availability of and the actual data being passed through the Port. The same mechanism that is used for transmitting data to a subroutine is used for Ports. Thus, the data passed is a pointer to an actual parameter list, the contents of which are accessed by the receiver through a formal parameter list. Two Ports communicate using the commands CONNECT, DISCONNECT, SEND, RECEIVE, CONDITIONAL RECEIVE, and REQUEST. Terminals and physical devices are handled by connecting the Port to a Port in a device dependent system program for the terminal or physical device that transforms the communication into I/O commands appropriate for the device, and which then requests the supervisor to perform the I/O through the MONITOR Port. In *ISPL*, each job has a MONITOR Port which is used for all communication with the jobs monitor. Files are handled similarly, except that the determination of the program to which the connection should be made is based on user supplied routines (for each file type) to create, destroy, connect, disconnect, and communicate with that file type.

Balkovich et al. and Gettys [BLP85,Get84] describe Project Athena, an experiment at MIT to explore the potential uses of advanced computer technology in the university curriculum. Corporate sponsors are DEC and IBM. By the end of the project, MIT will have a network of 2000 high performance graphics work stations. The system will provide facilities that will make it possible to share information and to access data and programs from any computer. The communications network is implemented with multiple technologies and is based on a high-speed backbone network. There are three major problems involved in building large distributed systems:

1. Scale

2. Different machine architectures

3. Different operating systems

Project Athena uses the UNIX operating system which provides the foundation needed to port all applications to all types of workstations. A working hypothesis of the Project is that most scientific and engineering applications can usefully interact only when employing a small number (20–30) of data types (e.g., graphs, arrays, tables). Interfaces between diverse applications may be defined with common representations of these data types and methods for manipulating them. This motivates keeping the number of supported programming languages small.

Hailpern [Hai86a] suggests four ways to build a multiparadigm language.

1. Combine the syntax and semantics of several languages; e.g., Combine the syntax and semantics of Prolog, Lisp, and C into one language. An advantage is that users can start using the system quickly as they are already familiar with one component of the system. A disadvantage is the unintended side effects caused by the complex interactions of different semantics.

2. Add new structures to an existing language; e.g., adding objects and methods to Pascal.

3. Redefine an existing language in the light of new theoretical discoveries. This allows for corrections to be made to existing languages.

4. Start from scratch and build a new system. The advantages are consistency and elegance but the effort to attract a user community may be prohibitive.

Stefik et al. [SBK86] working at the Xerox Palo Alto research center are developing a multiparadigm system called the Loops knowledge programming system. The Loops system integrates the following paradigms:

1. Functional. A pure mathematical specification of the solution to a problem, eliminating the conventional von Neumann model of memory and variables; e.g., pure Lisp, Backus's FP language.

2. Rules-oriented. Specifying the constraints of the problem, rather than the algorithm for finding a solution; e.g., Prolog, OPS5

3. Object-oriented. Grouping data into objects or abstract data types, where each object (or class of objects) has a set of operations (methods) to manipulate the data stored in that object; e.g., Smalltalk, Simula, CLU.

4. Access-oriented. The specification of side-effects or demons attached to the manipulation of variables; e.g., an extension of Loops.

Researchers at the University of Texas at Austin [Kor86] have tried to apply the relational data model to the following:

1. User interfaces: the use of relational languages to access directories and mailboxes modeled as relations.

2. Design databases: where design environments such as CAD and software development are being modeled as relations.

These applications are not the traditional data processing applications to which the relational model was exclusively applied initially. One drawback to the naturalness of expression in Relational languages was the restriction to atomic domains (first normal form). Allowing set-value and record-value domains will make representations more compact and provide the user with an intuitive view of the data. Another drawback: only three kinds of operators (i.e. insert, delete, update) to modify tuples. Two approaches are suggested for additional semantic rich operations.

1. Relations with side-effects

    - A queue relation is defined for each non-relational operation
    - To print a file, a tuple representing the file is enqueued
    - The file is ultimately printed and the tuple dequeued

2. Operator relations. Relational expressions may be embedded within a host programming language (C, PL/I or COBOL) and a special call is used to execute the relational operations.

In the object-oriented view, both data and methods need to be expressed and the internal structure must be hidden from the users. The object is represented as a tuple with attributes for the data and a single set valued attribute for the methods. Thus, two paradigms may exist.

1. Operation-oriented programming: when the data is brought to an operation in the form of a queue relation.

2. Object-oriented programming: where a message is sent to an object to perform a method.

A scheme to translate between these two is used so that one may work with either.

According to Tu and Perlis [TP86], functional programming creates more compact programs. Functional programming maintains referential transparency which means that variables can take a single value within a given scope. Programs are thus easier to debug and maintain. Functional programs show more parallelism because of the absence of side effects. The non-functional language, APL, offers array processing capabilities that enable condensed programs and highly parallel computation. However, a major weakness of APL is its semantics: gotos and side effects. It uses dynamic binding instead of lexical binding. The authors have attempted a functional APL language called FAC that combines the strengths and eliminates the weaknesses of the two paradigms. FAC has the same syntax as APL but FAC has functional semantics - lexical binding and no side effects or gotos.

Jenkins et al. [JGM86] describe the programming language Nials (Nested Interactive Array Language) which supports several styles of programming including:

1. Imperative; e.g., assembly language, Basic. These languages have constructs closely related to the intruction sets found in a von Neumann architecture. They include commands such as assignment and branching. They can evaluate formulae involving arithmetic and logical expressions.

2. Procedural; e.g., Algol, Fortran, PL/I, Pascal, C, Euclid, and Ada. The procedural style of programming includes imperative programming facilities, but supplements them with an abstraction mechanism to build procedures that generalize the concepts of a command and an expression.

3. Applicative; e.g., Lisp, ISWIM, Lucid. The applicative style of programming uses function applications and recursive function definitions as the main means of computation.

4. Lambda-free; e.g., Backus's FP language. This style limits the use of functional mechanisms to two levels; i.e., functions on data and combining forms that construct functions from other ones.

5. Array-oriented; e.g., APL. This style uses array data structures as the values in the data domain and has operations that map these values as a whole.

6. Relational; e.g., Prolog. The programmer provides a description of the problem and an underlying interpreter deduces the solution based on some presumed semantics.

The term "style of programming" has been used in a generic sense and means the style associated with a particular paradigm of computing. The design of Nial has been influenced by the desire to provide a multiparadigm programming language, suitable for teaching various styles. Nial is an exampe of a programming language based on More's array theory [Mor79,Mor81]. In array theory all data objects are arrays. Operations are functions that map arrays to arrays. Transformers are functions that map operations to operations. In essence, array theory is similar to a typed lambda calculus that is limited to functional objects of order two. While Nial does not directly support relational or object-oriented styles, they may be embedded in it.

Hailpern [Hal86] showcases nine multiparadigm research projects.

1. Arctic: a functionl language for real-time control

2. C++: an imperative, object-oriented language

3. CaseDE design environments (imperative and specification)

4. Lore: object-oriented, set-based

5. Orient84/K: object-oriented, rule-based, access-oriented, parallel

6. Smallworld: imperative, object-oriented

7. Tablog: functional, rule-based

8. Algebraic specifications in Prolog: specifications, rule-based

9. Integrating functional and logic programming: functional, rule-based

Hayes et al. [HS87,HMS88,MHS86b] describe a mixed language programming system using distributed computing. A mixed language program or *MLP* is written in two or more programming languages. A *MLP* consists of components. Each component is composed of one or more procedures written in the same language called a host language. A *MLP* system consists of six parts.

1. *UTS Language:* It consists of two parts. The first part is a collection of types and type constructors; e.g., integer, float, array$(10, 5)$ of integer. The second part makes use of symbols to construct signatures that denote sets of types.

A typical use may be to describe a parameter in a signature whose actual argument type may vary from call to call.

2. *Language Binding:* Defined for each host language. They specify the mappings between the host language types and the *UTS* language types. Only some mappings may have an exact equivalence.

3. *Agent:* Each program component has an envelope process called an agent for the language binding. An agent has a set of outgoing routines to handle calls to procedures outside the component and to translate arguments from the host language into *UTS*. An agent has a set of incoming routines to handle invocations of procedures and to translate arguments from *UTS* into the host language. An agent uses inter-process communication to implement cross-language calls.

4. *Interface Specification:* A list of interfaces exported from a component together with a list of imported interfaces. These imported and exported interfaces are specified in the *UTS* language as signatures that describe the number and type of arguments of the given procedure. For example:

   export "procname" <export signature>

   import "procname" <import signature>

5. *MLP Translator:* Each host language has one *MLP* translator. Its basic function is to take a program component and produce object code. Besides, a *MLP* translator has other tasks.

   (a) insert code for the agent.

   (b) place export and import signatures at known locations in a file.

   (c) change calls to external component procedures into calls on the local agent.

6. *MLP Linker:* It performs static type-checking of arguments of the inter component calls. The *MLP* linker inserts prologue code and epilogue code into the main program component. Prologue code establishes inter-process communication links and distributes the processes among different machines. Epilogue code sends messages to terminate a process when the program has terminated.

Manweiler et al. [MHS86a] describe how to add a new language to the Berkeley UNIX implementation of the Mixed Language Programming system. Hayes et al. [HHS88] describe the integration of the object-oriented, distributed programming language Emerald into the *MLP* system.

CHAPTER III

A MODEL FOR MULTIPARADIGM SOFTWARE REUSE

World Interconnection Model

This section introduces a new model for software design and development. The model called "World Interconnection Model" is based on the "Possible Worlds Model" described by Kripke [Kri63]. This model attempts to provide structure to a system composed of multiparadigm routines. The structure will allow "used in" **ui** and "used instead of" **uio** relations between two worlds based on the concept of views.

A world interconnection model is made up of several worlds. If it is possible to specify a sub–problem as an abstract data–type then this abstract data–type can be associated with a world. $A = \{\alpha_1, \alpha_2, \ldots\}$ is a set of agents. An agent is an instantiation of a world. An agent considers the world of which it is an instance the real world. It considers another world as "possible" if it cannot distinguish this other world from the real one; i.e., if its view of the real world is OK in the other world. In the world interconnection model, a software system is broken down into worlds. Let $\Sigma = (S, \pi, \Psi)$ be a software system. $S = \{s_1, s_2, \ldots\}$ is the set of all worlds. Methods, method interfaces, and propositions are defined at each world.

Define each world $s_i = M_i \cup E_i \cup I_i \cup P_i$. M_i is the set of methods defined at world s_i. An example of a method would be an algorithm to implement a procedure or function for the stack abstract data-type described in Figure 2.

Each method has an interface that describes it. E_i is the set of interfaces describing methods defined at world s_i. An example of interfaces is given by the specification of procedures and functions for the stack abstract data-type (Figure 2). Some accessible methods may be from other worlds. World s_i has a set of interfaces I_i corresponding to methods it can access. Accessible methods may be used in a method at a world. This "used in" relation **ui** resembles the object-oriented paradigm client relation. Accessible methods may be used instead of methods defined at a world. This "used instead of" relation **uio** corresponds to the object-oriented paradigm inheritance relation.

P_i is the set of propositions defined at s_i. Propositions may be modal formulas such as $\Box p$ or $\Diamond p$. Language, communication, and architecture specifications may be included in propositions. This information may be used during design, compilation, and maintenance.

Worlds may be dormant or instantiated. An example of a dormant world is an abstract data-type description of a stack while an instantiation is an execution of an instance of a stack abstract data-type also called an agent. Worlds may also be classified as abstract and concrete. An abstract world has no defined methods; i.e., $M = \{\}$. For a concrete world, $M \neq \{\}$.

- procedure Push(var S: stack; x: T);

  pre: $\sim$ full(S)

  post: S = x $\sim$ S$'$;

- function Pop(var S: stack): T;

  pre: $\sim$ empty(S)

  post: S$'$ = RESULT $\sim$ S;

- function Top(S: stack): T;

  pre: $\sim$ empty(S);

  post: RESULT = first(S);

- procedure Clear(var S: stack);

  post: S = <>;

- function Empty(S: stack): boolean;

  post RESULT $\equiv$ (S = <>);

- function Full(S: stack): boolean;

  post: RESULT $\equiv$ (length(S) = maxstack);

Figure 2. Specification of a stack ADT [WSHF81]

Every world has a set of views associated with it. Let $V_\imath$ be the set of all views possible at world s_i. Define $V_\imath = 2^{E_\imath \cup I_\imath \cup P_\imath}$; i.e., V_i is the set of all subsets of the present and accessible method interface, and proposition sets. An agent α has view v_α of world $s_\imath$. v_α is defined as a set of method interfaces and formulas that are well–defined at that world. An agent α may choose or be assigned a view v_α of a world. If an agent is allowed to choose its view of the world then $v_\alpha \in 2^{E_\imath \cup I_\imath \cup P_\imath}$. If an agent is assigned a view of the world then $v_\alpha \in X$, $X \subset 2^{E_\imath \cup I_\imath \cup P_\imath}$. For a stack abstract data-type, a view may be a subset of the method interfaces [WSHF81] described in Figure 2.

$\Psi = \{\Psi^1, \Psi^2, \ldots\}$ is the set of all possibility relations between worlds. With view v_α, agent α may develop possibility relation Ψ_α. An ordered pair $(s_1, s_2) \in \Psi_\alpha$ if agent α's view v_α of world s_1 is OK at world s_2; i.e., if agent α considers world s_2 a possible world when its view of world s_1 is v_α. Since different views may exist for the same world, different possibility relations can develop with different worlds. To define Ψ_α, view v_α needs to be OK at world s. Define π as a function from the set $S \times V \mapsto \{OK, \neg OK\}$. Function π either $OK's$ or $\neg OK's$ a view at a world. A view v_α is OK at a world s if all the interfaces ι in the view are supported by methods m from the set of methods M defined at world s and if all propositions φ in the view are *true* at world s.

<u>Definition 3.1</u>

$$\pi(s, v_\alpha) = OK \ if \ (\forall \iota \in v_\alpha, \ \iota \mapsto m, \ m \in M) \bigwedge (\forall \varphi \in v_\alpha, \ \varphi \mapsto true)$$

M is a set of methods at s.

A view v_α is not OK at a world s if there exists atleast one interface ι in the view that is not supported by any method m from the set of methods M defined at world s or if there exists atleast one proposition φ in the view that is *false* at world s.

Definition 3.2

$$\pi(s, v_\alpha) = \neg OK \ if \ (\exists \iota \in v_\alpha,\ \iota \not\mapsto m,\ m \in M) \bigvee (\exists \varphi \in v_\alpha,\ \varphi \mapsto false)$$

As an illustrative example, consider the Stack and Deque ADT's. Stack may consider deque as a possible world because the stack view is OK at deque. Therefore, it is possible to introduce a meaningful relation between stack and deque that relates to the world interconnection model. The world interconnections can be represented using a directed graph. Nodes represent worlds and arcs represent possibility relations. A possibility relation from stack to deque is indicated using an arrow from stack to deque. (Figure 3).

For a queue abstract data-type, a view may be a subset of the method interfaces [WSHF81] in Figure 4. This view is also OK at deque. So, a possibility relation may exist from queue to deque (Figure 3). Thus two different abstract data-types, stack and queue, regard deque as a possible world and they have two completely different views of deque. This structure intuitively mimics the relation between a class (deque) and the subclasses (stack and queue) as defined in one model of object-oriented inheritance [HN87]. For a deque abstract data-type, a view may be a subset of the method interfaces [WSHF81] described in Figure 5 and 6.

Can an agent that considers deque as the real world consider either stack or

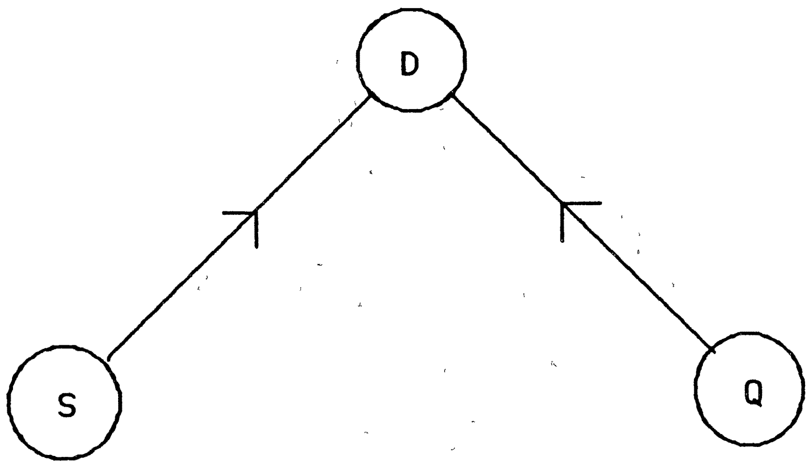

Figure 3. Possibility relations from stack & queue to deque

queue as a possible world? Yes, if its view of deque is OK in the stack or queue worlds. This arguement is more readily acceptable if you consider the many possible values that a view may have. This constitutes a mutual relationship in which worlds access methods defined at each other (Figure 7).

Now suppose deque considers table as a possible world. This does not necessarily mean that agents at stack and queue automatically consider table as a possible world. This is because the possibility relation between deque and table has been formed through the deque view. This view is different from the stack and queue views. The agents at stack and queue will have to independantly establish possibility relations with table. This allows for a flexible model of structure.

- procedure Nq(var Q: queue; x: T);

  pre: $\sim$ full(Q)

  post: $Q = Q' \sim x$;

- function Dq(var Q: queue): T;

  pre: $\sim$ empty(Q)

  post: $Q' = \text{RESULT} \sim Q$;

- function Peek(Q: queue): T;

  pre: $\sim$ empty(Q);

  post: RESULT = front(Q);

- procedure Clear(var Q: queue);

  post: Q = <>;

- function Empty(Q: queue): boolean;

  post RESULT $\equiv$ (Q = <>);

- function Full(Q: queue): boolean;

  post: RESULT $\equiv$ (length(Q) = maxqueue);

Figure 4. Specification of a queue ADT [WSHF81]

- procedure Push(var D: deque; x: T);

  pre: $\sim$ full(D)

  post: D = x $\sim$ D$'$;

- procedure Nq(var D: deque; x: T);

  pre: $\sim$ full(D)

  post: D = D$'$ $\sim$ x;

- function Pop(var D: deque): T;

  pre: $\sim$ empty(D)

  post: D$'$ = RESULT $\sim$ D;

- function Dq(var D: deque): T;

  pre: $\sim$ empty(D)

  post: D$'$ = D $\sim$ RESULT;

- function Front(D: deque): T;

  pre: $\sim$ empty(D);

  post: RESULT = first(D);

- function Rear(D: deque): T;

  pre: $\sim$ empty(D);

  post: RESULT = last(D);

Figure 5. Specification of a deque ADT (first part)

- procedure Clear(var D: deque);

  post: D = <>;

- function Empty(D: deque): boolean;

  post RESULT ≡ (D = <>);

- function Full(D: deque): boolean;

  post: RESULT ≡ (length(D) ≥ maxdeque);

Figure 6. Specification of a deque ADT (second part) [WSHF81]

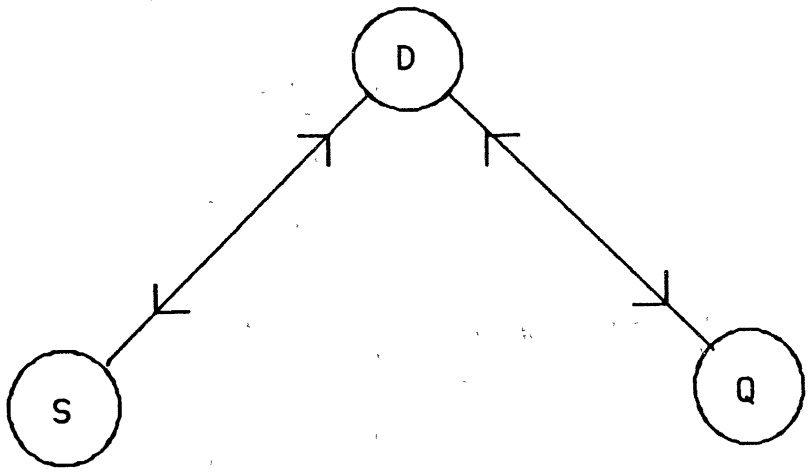

Figure 7. Possibility relations between stack, queue, & deque

In the hierarchical model (Figure 8), *Stack* and *Queue* inherit from *Deque*, and *Deque*, *Stack* and *Queue* inherit from *Table*. If some change is made to T that is acceptable to S but not to Q then the hierarchy relations are compromised. In the world interconnection model, S and Q have views that are OK at D and T. D has a view that is OK at T. This results in the formation of possibility relations Ψ^{SD}, Ψ^{QD}, Ψ^{ST}, Ψ^{QT}, Ψ^{DT} (Figure 8). Any change made at T may affect the possibility relations Ψ^{ST}, Ψ^{QT}, Ψ^{DT}. These changes will not affect possibility relations Ψ^{SD}, Ψ^{QD}.

There may be a main program world containing a main program method. The main program may contain subroutine calls or invocations to methods in other worlds. A call or invocation has an interface and these interfaces may be incorporated into views. Based on views, a possibility relation may be formed with another world. This concludes a description of the static structure of the system.

Execution may begin at the main program or it may begin with a concurrent instantiation of several agents. Inter agent communication takes place based on which views are in use.

An Example

Integer, real, and complex numbers are represented by three worlds i, r, and c respectively. Following our established notation, M is a set of methods, E the corresponding interfaces, and I a set of interfaces of methods to be "used in" or "used instead of".

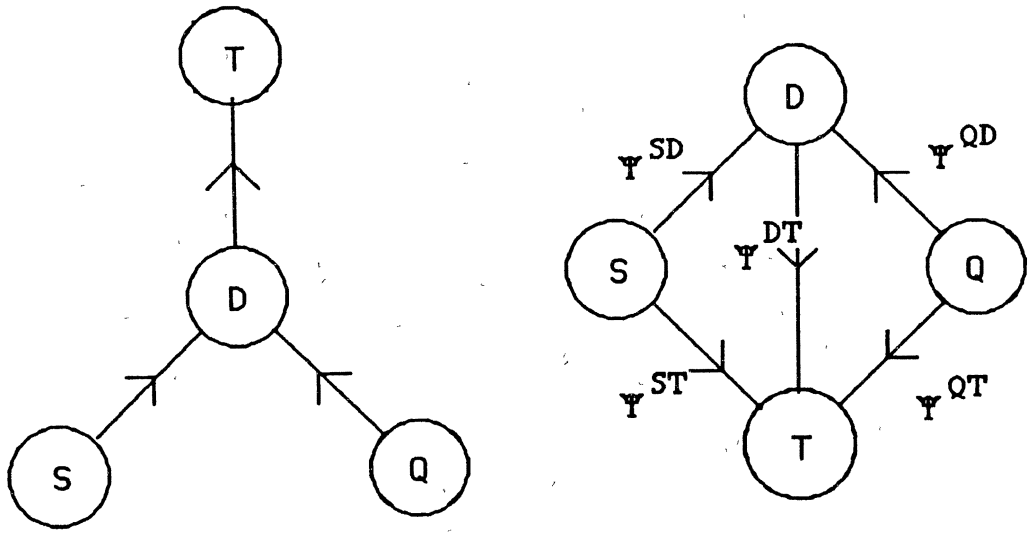

Figure 8. Hierarchical & world interconnection models

$M_i = \{+_i\}$, $E_i = \{var_i = exp_i\ oper_i\ exp_i\}$

$M_r = \{+_r\}$, $E_r = \{var_r = exp_r\ oper_r\ exp_r\}$

$M_c = \{+_c\}$, $E_c = \{var_c = exp_c\ oper_c\ exp_c\}$

Figure 9 gives the possibility relations between worlds in the sense of "used instead of" **uio**. Therefore, $+_r$ can be used instead of $+_i$ at world i; $+_c$ can be used instead of $+_i$ and $+_r$ at worlds i and r respectively. The reflexive relations at worlds i, r, and c are **ui** "used in" relations. $+_i$, $+_r$ and $+_c$ are used in the respective worlds i, r, c.

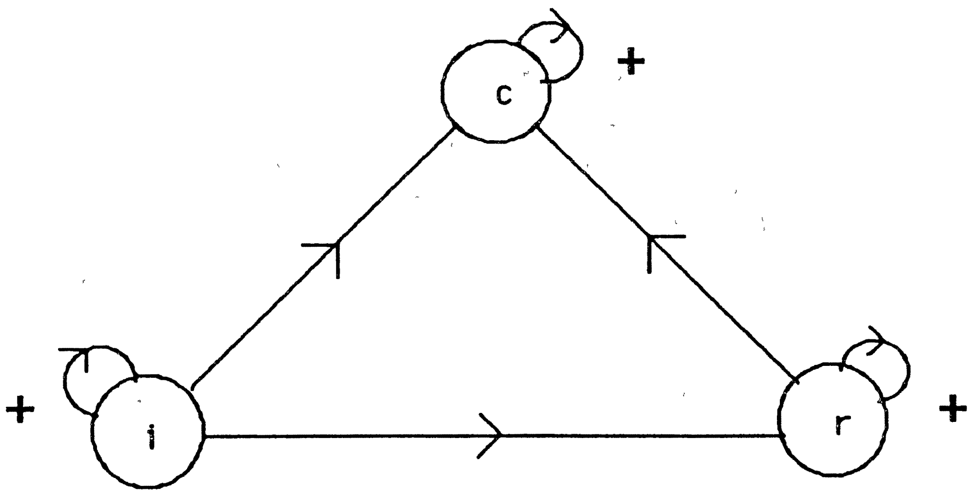

Figure 9. Relations between i, r, c with view +

The operations of complex subtraction $-_c$, complex multiplication $*_c$, and complex division $/_c$ may be defined in terms of real operations.

$$(w, x) -_c (y, z) = (w -_r y, x -_r z)$$

$$(w, x) *_c (y, z) = (w *_r y -_r x *_r z,\ x *_r y +_r w *_r z)$$

$$\frac{(w,x)}{(y,z)} = \frac{(w*_r y +_r x*_r z,\ x*_r y -_r w*_r z)}{y^2 +_r z^2}$$

Figure 10 gives the possibility relations between worlds. The relation from r to c illustrates the "used instead of" **uio** relation while the relation from c to r illustrates the "used in" **ui** relation and "used instead of" **uio** relation.

A possibility relation from c to r allows $+_r, -_r, *_r, /_r$ to be used in a method

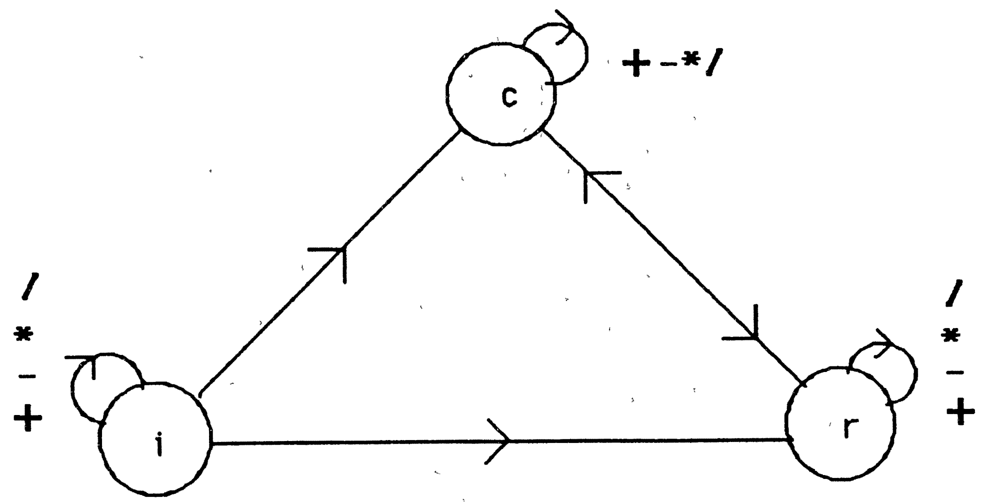

Figure 10. **ui, uio** relations between i, r, c with view (+ - * /)

defined at c. Alternately, the possibility relation allows for methods defined at r corresponding to $+_c, -_c, *_c, /_c$ to be used instead of these at c. For world i, the operations $+_r, -_r, *_r, +_c, -_c, *_c$ may be used instead of the corresponding integer operations. $/_r$ and $/_c$ may be used in some method that duplicates the $/_\imath$ operation.

An Introduction to Modal Logic

Necessity, impossibility, contingency and possibility are modal notions and modal logic is the logic of these notions. Chellas [Che80] defines modal logic as the logic of necessity and possibility. According to Konyndyk [Kon86], a modal logic should provide a way to exhibit the logical structure of those inferences that use modal concepts in a way which affects their validity. The following symbols are commonly used to represent the modal operators.

1. $\Box p$: "p is necessary." Konyndyk [Kon86] defines a necessity as that whose denial is self inconsistent. Examples of necessity are:

   (a) $\forall p, q,\ if\ p = true \wedge p \rightarrow q,\ then\ q = true$

   (b) $7 + 5 = 12$

   (c) $A \cup B = B \cup A$

   The other modal operators may be defined in terms of necessity.

2. $\Diamond p$: "p is possible." A proposition p is possible in case it is not necessary that it be false. $\Diamond p = \neg\Box\neg p$

3. $|p$: "p is impossible." A proposition p is impossible in case it is necessary that it be false. $|p = \Box\neg p$

4. A contingent proposition is one that is both possibly true and possibly false.

5. Two propositions are consistent or compatible iff their conjunction is possible.

6. Two propositions are inconsistent or incompatible iff their conjunction is impossible.

7. Two propositions are contradictory iff both cannot be true and both cannot be false.

Historical Note

Lemmon and Scott [LS77] have summarized the historical development of modal logics. Aristotle explored the modal notions in his "De Interpretation" ch. 12 and ch. 13. He arrived at the following results:

1. $\Box A$ and $\neg\Box A$ is a contradiction.

2. $\Diamond A$ and $\neg\Diamond A$ is a contradiction.

3. $\Box A$ and $\Box\neg A$ is not a contradiction.

4. $\Diamond A$ and $\Diamond\neg A$ is not a contradiction.

Aristotle detected two senses of the possibility operator:

1. $\Box A \rightarrow \Diamond A$ which is possibility proper.

2. $\Diamond A \rightarrow \Diamond\neg A$ eg. contingency.

Aristotle gave two equivalences for impossibility:

1. $|A \equiv \Box\neg A$

2. $|A \equiv \neg\Diamond A$

From these we may derive the famous equivalences:

1. $\Diamond A = \neg\Box\neg A$

2. $\Box A = \neg\Diamond\neg A$

The Megarians and the Stoics developed theories concerning modality that suggest a connection between modalities and temporal notions. This connection is supported by recent work on the logic of time. They defined the possible as that which either is or will be; the impossible as that which being false, will not be true; the necessary as that which being true, will not be false; and the non-necessary as that which either is already or will be false.

In the middle ages, Pseudo-Scotus added these variations:

1. $\neg\Diamond A = \Box\neg A$

2. $\neg\Box A = \Diamond\neg A$

He also considered modes such as "to doubt, to know, to wish" and pointed out similarities between these and the usual modalities. Recent work in epistemic logic corroborates his observations.

Contemporary Work

Modern work in modal logic is attributed to C. J. Lewis. Lewis tried to define an implication without the paradoxes of Russel and Whitehead in "Principia Mathematica." In "Symbolic Logic", Lewis and Langford defined strict implication $\Rightarrow$ in this way.

$$A \Rightarrow B = \neg(\Diamond A \wedge \neg B)$$

This definition of strict implication, also called entailment, expresses the fact that a conditional is true precisely when the negation of the consequent is incompatible with the antecedent. Lewis intended that if A entails B then B should follow logically from A. This definition led to the following paradoxes:

1. Anything whatsoever follows from the impossible

$$\neg\Diamond A \rightarrow (A \Rightarrow B)$$

2. The necessary follows from anything whatsoever

$$\Box B \rightarrow (A \Rightarrow B)$$

Lewis accepted these paradoxes because they are the consequence of acceptable and straightforward rules that are in everyday use. Lewis describes five systems of modal logic, $S1 - S5$ each with its own rules and theorems.

Structure and Modality

Different systems of modal logic are described by Konyndyk [Kon86]. Each has different theorems and validates different inferences. Saul Kripke suggested the idea of possible worlds and possibility relations between worlds as a way of looking at differences between modal systems. In this section we discuss applications of different systems of modal logic to the world interconnection model.

Read $s \models v$ as "view v is OK at world s.". In the formulas below, s_1 and s_2 are not necessarily distinct.

World s_1 considers world s_2 a possible world if a view of s_1 is OK in s_2.

$$(s_1, s_2) \in \Psi \ if \ \exists v \in V_{s_1}, (s_2 \models v)$$

A view v is possible in a world s_1 if it is OK in some world related to s_1 by a possibility relation Ψ.

$$s_1 \models \Diamond v \ if \ \exists s_2, (s_1, s_2) \in \Psi, s_2 \models v$$

A view v is necessary in a world s_1 if it is OK in every world related to s_1 by a possibility relation Ψ.

$$s_1 \models \Box v \ if \ \forall s_2, (s_1, s_2) \in \Psi, s_2 \models v$$

In the rest of this section, the modal systems $T, X, S4, S5$ are applied to the world interconnection model using as example the worlds $stack, queue$ and $deque$.

The modal system T has the characteristic formula $\Box\varphi \supset \varphi$; i.e., whatever is logically necessary is OK. If the possibility relations at a world in a possible worlds model are reflexive, the characteristic formula of T holds. A view that includes the formula $\Box\varphi$ requires φ to be defined (for a method) or true (for a proposition) at that world. In Figure 11, view $v_{stack} = \Box push$. By definition, $\Box push$ must be OK in all worlds possible from $stack$; i.e., $\Box push$ must be OK at $stack$ and $deque$. Since the relation at $stack$ is reflexive, the the characteristic formula of T, $\Box push \supset push$ must hold at $stack$. Thus

$\Box push \supset push$

$\underline{\Box push \qquad\qquad}$

$push$ (modus ponens)

Therefore $push$ must be defined at $stack$. Since $\Box push$ holds at $stack$, by definition, $push$ must be defined at $deque$.

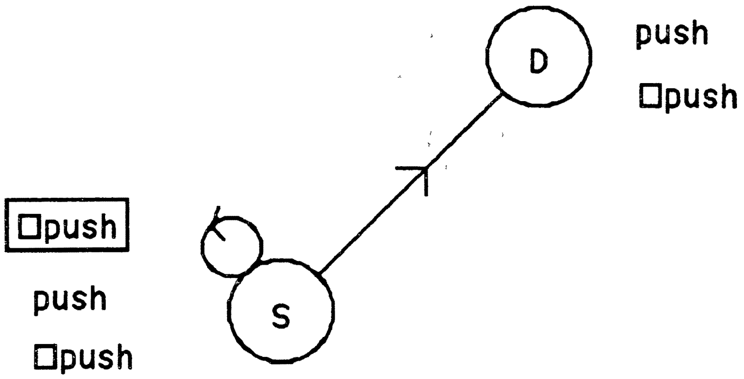

Figure 11. Reflexive relations at worlds

The system X has the characteristic formula $\varphi \supset \Box\Diamond\varphi$. If the possibility relations at a world are reflexive and symmetric, the characteristic formula of X holds. A view that includes the formula φ requires $\Box\Diamond\varphi$ to be true at that world.

In Figure 12, view $v_{stack} = push$. By definition, *push* must be *OK* in all worlds possible from *stack*; i.e., *push* must be *OK* at *stack* and *deque*. Since the relation at *stack* is reflexive and symmetric, the the characteristic formula of X, $push \supset \Box\Diamond push$ must hold at *stack*. Thus

$push \supset \Box\Diamond push$

$\underline{push \qquad\qquad\qquad}$

$\Box\Diamond push$ (modus ponens)

Therefore $\Box\Diamond push$ must hold at *stack*. Since the relation at *stack* is reflexive, the characteristic formula of T, $\Box(\Diamond push) \supset \Diamond push$ must hold at *stack*. Thus

$\Box\Diamond push \supset \Diamond push$

$\underline{\Box\Diamond push}$

$\Diamond push$ (modus ponens)

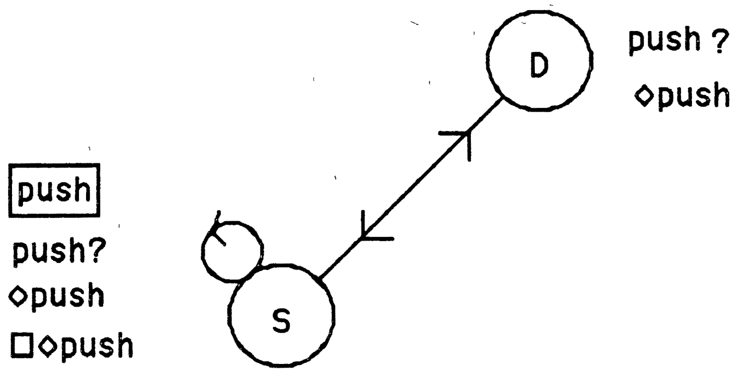

Figure 12. Reflexive & symmetric relations at worlds

Therefore $\Diamond push$ must hold at *stack*. Since $\Box\Diamond push$ holds at *stack*, by the definition of $\Box$, $\Diamond push$ must hold at *stack* and *deque*. By the definition of $\Diamond$, it is sufficient that *push* be defined at either *stack* or *deque*.

The system $S4$ has the characteristic formula $\Box\varphi \supset \Box\Box\varphi$. If the possibility relations at a world are reflexive and transitive, the characteristic formula of $S4$

holds. A view that includes the formula $\Box\varphi$ requires $\Box\Box\varphi$ to be true at that world.

In Figure 13, view $v_{stack} = \Box push$. By definition, $\Box push$ must be *OK* in all worlds possible from *stack*; i.e., $\Box push$ must be *OK* at *stack*, *deque* and *table*. Since the relation at *stack* is reflexive and transitive, the the characteristic formula of *S*4, $\Box push \supset \Box\Box push$ must hold at *stack*. Thus

$\Box push \supset \Box\Box push$

$\underline{\Box push \qquad\qquad}$

$\Box\Box push$ (modus ponens)

Therefore $\Box\Box push$ must hold at *stack*. Since the relation at *stack* is reflexive, the characteristic formula of *T*, $\Box(\Box push) \supset \Box push$ must hold at *stack*. Thus

$\Box\Box push \supset \Box push$

$\underline{\Box\Box push \qquad\qquad}$

$\Box push$ (modus ponens)

Therefore $\Box push$ must hold at *stack*. Since $\Box\Box push$ holds at *stack*, by the definition of $\Box$, $\Box push$ must hold at *deque*..

Since the relation at *stack* is reflexive, the characteristic formula of *T*, $\Box push \supset push$ must hold at *stack*. Thus

$\Box push \supset push$

$\underline{\Box push \qquad\qquad}$

$push$ (modus ponens)

Therefore *push* must hold at *stack*. Since $\Box push$ holds at *stack*, by the definition of $\Box$, *push* must hold at *deque*.

At *deque*, $v_{deque} = \Box push$. By definition, $\Box push$ must be *OK* at all worlds posible from *deque*; i.e., $\Box push$ must hold at *table*. From the definition of $\Box$, it is necessary that *push* be defined at *table*. The definition of *push* and the truth of $\Box push$ at *table* is also justified by the transitive relation from *stack* to *table*.

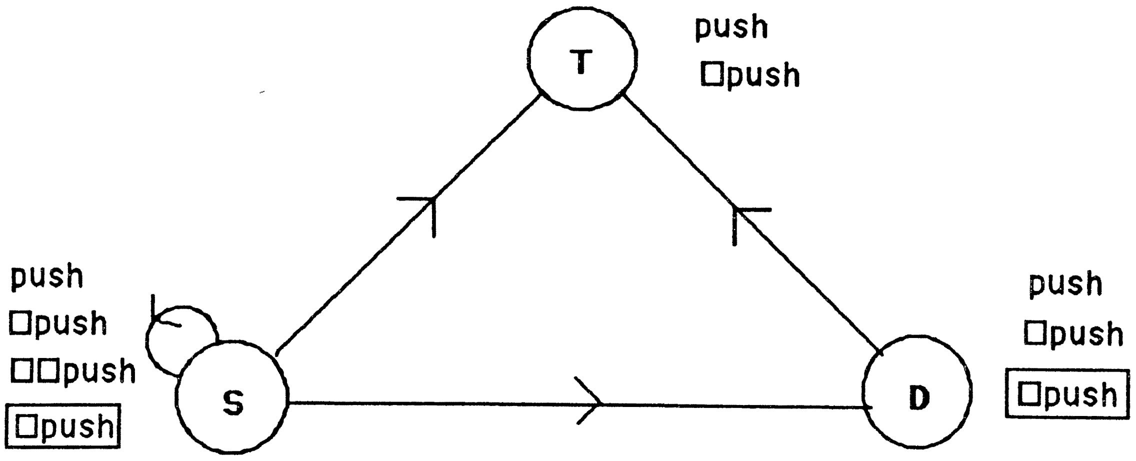

Figure 13. Reflexive & transitive relations at worlds

The system $S5$ has the characteristic formula $\Diamond\varphi \supset \Box\Diamond\varphi$. If the possibility relations at a world are reflexive, symmetric and transitive, the characteristic formula of $S5$ holds. A view that includes the formula $\Diamond\varphi$ requires $\Box\Diamond\varphi$ to be true at that world.

In Figure 14, view $v_{stack} = \Diamond push$. By definition, $\Diamond push$ must be *OK* in all worlds possible from *stack*; i.e., $\Diamond push$ must be *OK* at *stack*, *deque* and *table*.

Since $\Box\Diamond$*push* holds at *stack*, applying the definition of $\Box$ we have $\Diamond$*push* true at *stack*, *deque* and *table*. From the definition of $\Diamond$, *push* may be defined at *stack*, *deque*, *table*, or any possible world of *deque* or *table*.

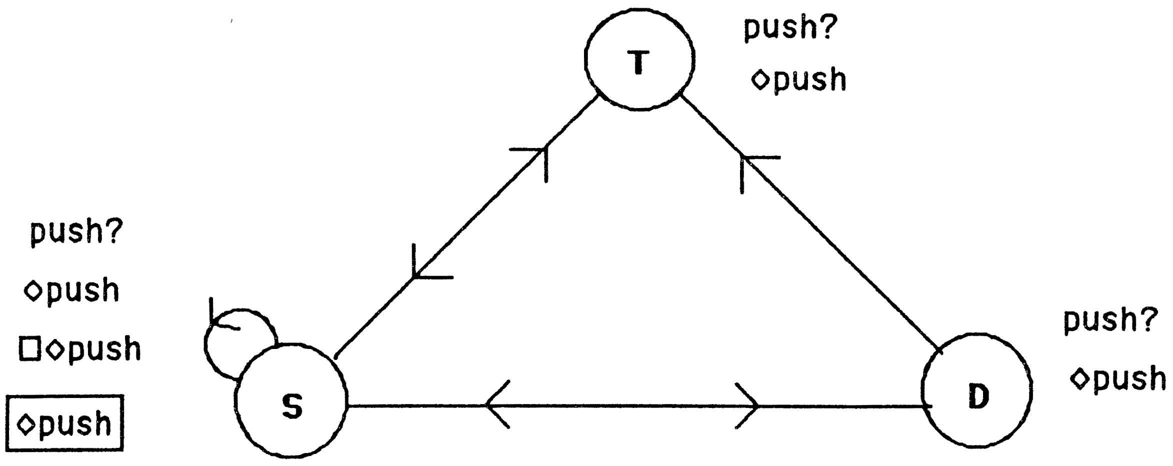

Figure 14. Reflexive, symmetric & transitive relations at stack

The immediate question is "How may all this be used?" In CHAPTER IV, a language specification corresponding to the model is defined. The compiler for such a language may establish a possibility relations graph. From such a graph, the compiler can distinguish which modal systems are present. The corresponding modal formulas that hold are established. Consistency questions about the system may be answered based on these formulas. The questions addressed in the previous discussion provide a framework for a general application of modality.

The possible worlds model, when combined with a modal logic, allows a user to reason about its knowledge. A user knows a formula if that formula is true in all worlds it considers possible. But what is so important about what a user knows? We have already stated that the possible worlds model establishes a certain structure on the system. This structure is based on the concept of a view. If this view changes, then the worlds that are possible may change because a possibility relation may not hold. Thus, the system is dynamic.

To know a formula means to establish the truth of the formula in all possible worlds. This is written as

$$\models K\varphi$$

A formula may be true in atleast one possible world. We write this as

$$\models \Diamond\varphi$$

By asking questions about what is possible, a user can identify those possible worlds that are useful to it. These questions may be as explicit as "Is a Prolog routine available for the stack abstract data-type?"

Thus, from a partitioning of the system into worlds based on abstract specifications, we have created structure. This enables a user to get replies to questions on necessity and possibility and this allows for the construction of extendible, multiparadigm systems.

CHAPTER IV

THE LANGUAGE AND ITS DESCRIPTION

A model for software reuse in a multiparadigm environment is of little practical value without a compatible language. A problem may be described in a way compatible with the model, using a "World Interconnection Language." A specification for such a language is given below. An informal description of the semantics of the phrase structures is provided. BNF notation is used to describe production rules. Non-terminals (representing concepts) are enclosed in < > brackets.

World Interconnection Language

$$< world\_system >::= < world\_system >< module > \mid < module >$$

The model describes a system as composed of several modules that may be of two different types.

$$< module >::= < world\_module > \mid < agent\_module >$$

The system has two kinds of modules. A *world_module* is an uninstantiated world description while an *agent_module* specifies instances of several agents. Each agent is an instantiated *world_module*.

$$< world\_module >::= \ world < world\_name >: \{$$
$$< method\_list >< interface\_list >< proposition\_list >\}$$

A *world_module* corresponds to a world in the model. It specifies a collection of available methods, a list of interfaces to be used in or used instead of existing methods, and a list of propositions that are OK at the world. Any two *world_modules*, though belonging to the same system, are independant of each other. This avoids any committment to object-oriented style client and inheritance relations in the external interface of a *world_module*. Thus, the system is reusable and extendible. This aspect of the model is discussed and illustrated in Example 2.

$$< agent\_module >::= \ create < agent\_module\_name >: \{$$
$$< agent\_list >< possrel\_list >\}$$

An *agent_module* corresponds to a system of instantiated agents.

$$< method\_list >::= \ implementation : \{< implementation\_spec\_list >\}$$
$$|\ comment : \{<< any\_text >>\}$$

The *method_list* specifies the path of a file_name containing the source code of the method. The *comment* statement includes text for descriptive or debugging purposes.

$$< interface\_list >::= \ < interface\_list >< interface > \mid < interface >$$
$$< proposition\_list >::= \ < proposition\_list >< proposition > \mid < proposition >$$
$$< proposition >::= \ < debug\_query > \mid < system\_info >$$

Remember that a view may consist of a list of interfaces and propositions. A view containing interfaces is used to construct possibility relations. A view may contain a debug query that traces an execution of an agent. In Example 2, modal operators are introduced to guide the compiler.

$$
\begin{aligned}
< agent\_list >::= \quad & < agent\_list > agent < agent\_name >:< world\_name > \\
& \mid agent < agent\_name >:< world\_name > \\
< possrel\_list >::= \quad & < possrel\_list > possrel < agent\_name > \\
& < world\_view >< world\_list > \\
& \mid possrel < agent\_name >< world\_view >< world\_list >
\end{aligned}
$$

The *agent_list* is a list of agents that are instances of some world. An agent is associated with a *world_view*. This world view may be satisfied in any of the worlds in the *world_list*. This corresponds to the formation of a possibility relation. The *world_list* specifies an agent name and interface names. If sufficient semantics are present in a world, the *world_list* may only contain agent names. The interface description combines the work of Hayes et al. and Purtilo [HS87,Pur86].

$$< interface >::= \quad < interface\_name >: \{< interface\_spec >\}$$

$$< interface\_spec >::= \quad < type\_spec > \mid < record\_spec >$$

$$
\begin{aligned}
< \mathit{type\_spec} >::= \; & < \mathit{type\_name} > \\
& \mid < \mathit{type\_spec} > (*) \mid < \mathit{type\_spec} > (+) \\
& \mid < \mathit{type\_spec} > (<< \mathit{integer\_value} >>) \\
& \mid < \mathit{type\_spec} >; < \mathit{type\_spec} > \\
& \mid < \mathit{type\_spec} >, < \mathit{type\_spec} >
\end{aligned}
$$

$$< \mathit{record\_spec} >::= \; \mathit{record} < \mathit{record\_name} >: \{< \mathit{type\_spec} >\}$$

$$< \mathit{type\_name} >::= \; \mathit{int} \mid \mathit{real} \mid \mathit{char} \mid \mathit{boolean}$$

The set of $< \mathit{type\_name} >$ defined herein contains the fundamental types integer, real, char and boolean. This may be modified to include other types or objects. The semi–colon denotes a sequential relationship between arguments while the comma operator denotes an alternation relation.

Repeated elements as in lists or arrays are indicated by a parenthesized integer, asterisk, or plus. The exact number of components is indicated by an integer. A non–negative number of components is indicated by an asterisk. A plus sign indicates a positive number of components. An interface may also contain a collection of $< \mathit{type\_spec} >$ in a $< \mathit{record\_spec} >$.

$$
\begin{aligned}
< \mathit{world\_list} >::= \; & < \mathit{world\_list} >< \mathit{agent\_name} >< \mathit{world\_view} > \\
& \mid < \mathit{world\_list} >< \mathit{agent\_name} > \\
& \mid < \mathit{agent\_name} >< \mathit{world\_view} > \\
& \mid < \mathit{agent\_name} >
\end{aligned}
$$

$$< \mathit{world\_view} >::= \; < \mathit{interface\_names} > \mid < \mathit{propositions} >$$

Example 1: Multiparadigm Software Reuse

This example, first presented by Purtilo [Pur86], is an illustration of software reuse in a multiparadigm environment. The world interconnection language is used to partition the system into worlds. Instantiated worlds called agents are created and possibility relations between agents are developed by binding interfaces.

Some lisp application requires the generation of a cubic spline to interpolate data. The data is represented as a list of ordered pairs.

$$((x_1 y_1)(x_2 y_2) \ldots (x_n y_n))$$

x_i and y_i represent the abscissa and ordinate respectively of any one data point. A fortran library routine paraphrased in Figure 15 is used.

If the spline routine is being invoked from a main lisp routine then the problem may be represented using the world interconnection language. Each of the two routines is identified as a world. The implementations are in files. The *spline* view of world *userlisp* sends a list of tuples and accepts a list of triples. World *fmmobject* also has a view called *spline*. This interface accepts an integer number representing the number of points and two arrays of points. It returns three arrays of points. The world description is given below.

$$\begin{array}{ll} world\ userlisp : \{ & implementation : \{/v/bren/prog1.lisp\} \\ & spline : \{\{float, float\}(*), \{float, float, float\}(*)\}\} \\ world\ fmmobject : \{ & implementation : \{/v/bren/prog2.fortran\} \\ & spline : \{int, float(*), float(*), float(*), float(*), float(*)\}\} \end{array}$$

```
subroutine spline(n, x, y, b, c, d)
integer n
double precision x(n), y(n), b(n), c(n), d(n)
The coefficients b(i), c(i), and d(i), i = 1, 2, ..., n
are computed for a cubic interpolating spline
$s(x) = y(i) + b(i) * (x - x(i)) + c(i) * (x - x(i))^2 + d(i) * (x - x(i))^3$
for $x(i) \leq x \leq x(i+1)$
input
n = the number of data points ($n \geq 2$)
x = abscissas of points in strictly increasing order
y = ordinates of the points
output
b, c, d = arrays of spline coefficients as defined above
```

Figure 15. Paraphrased excerpt from library

Let L and F be agents of the respective worlds. Using view *spline* of agent L, a possibility relation is constructed to agent F.

$$\begin{aligned} create\ example : \{ \ & agent\ L : userlisp \\ & agent\ F : fmmobject \\ & possrel\ L\ spline\ F\ spline\} \end{aligned}$$

As things stand, the interfaces do no mesh since the argument patterns do no match. What is necessary is another routine to perform coercion as shown below.

$$\begin{aligned} world\ newrule : \{ \ & implementation : \{/v/bren/prog3.pascal\} \\ & input : \{\{float, float\}(*), \{float, float, float\}(*)\} \\ & output : \{int, float(*), float(*), float(*), float(*), float(*)\}\} \\ create\ example : \{ \ & agent\ L : userlisp \\ & agent\ F : fmmobject \\ & agent\ N : newrule \\ & possrel\ L\ spline\ N\ input \\ & possrel\ N\ output\ F\ spline\} \end{aligned}$$

Example 2: Exposure of Inheritance

Alan Snyder [Sny87] questions whether or not the use of inheritance in the construction of a software component should be exposed to clients (inheriting or instantiating). Object-oriented inheritance allows a subclass to inherit methods from any ancestor. A client of a subclass may access instance variables of an ancestor class. Consequently, a change to a variable name or a method in any ancestor class will affect all dependant subclasses and clients. In the world interconnection model connections between world modules are defined using the world interconnection language which defines possibility relations between independant modules using views. The possibility relations relate one world to another and no further. Views allow access to allowed methods only. All variables may be accessed only through interfaces. Changes to any method will affect all worlds that access that method through a possibility relation.

Consider the deque and stack abstract data–types defined in CHAPTER III. Deque and stack have operations push, pop, top, and empty. In addition, deque has the operations nq, dq, front. In order to implement the abstraction defined in Figure 16, define the class stack to inherit (in the sense "use instead of") operations from the class deque, ignoring the additional operations.

A self-contained definition of the operations may also be included for stack. Clients of stack should be able to switch between the inherited and self-contained definitions. However, in some cases the absence of the extra operations nq, dq, and front could lead to an application break–down. This occurs when some client of stack, using the operations defined at deque, starts to use the operations defined at a stack, at which the extra operations that it used are missing. It is necessary to exclude the extra operations from the external interface of stack. This exposure of inheritance introduces a dependancy between stack and deque. The designer is not free to change the inheritance hierarchy without affecting clients of stack.

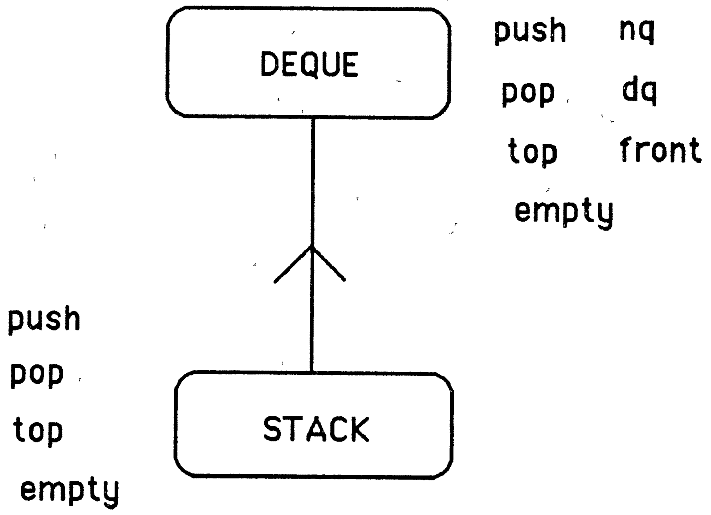

Figure 16. Implementing stack using deque

This exposure of inheritance is avoided using the world interconnection language. The view of the stack world does not include the operations nq, dq, and front. Thus the possibility relations between stack and deque is formed from a view that excludes these extra operations. No client of stack can use these extra operations because the view will not allow it. An illustrative implementation is given below.

$world\ deque : \{\{\ \ implementation : \{/v/bren/push.pas\}$

$push\{real, real(*)\}\}$

$\{implementation : \{/v/bren/pop.pas\}$

$pop\{real, real(*)\}\}$

$\{implementation : \{/v/bren/top.pas\}$

$top\{real, real(*)\}\}$

$\{implementation : \{/v/bren/empty.pas\}$

$empty\{boolean, real(*)\}\}$

$\{implementation : \{/v/bren/nq.pas\}$

$nq\{real, real(*)\}\}$

$\{implementation : \{/v/bren/dq.pas\}$

$dq\{real, real(*)\}\}$

$\{implementation : \{/v/bren/front.pas\}$

$front\{real, real(*)\}\}\}$

$world\ stack : \{\{\ \ implementation : \{/v/bren/push.pas\}$

$push\{real, real(*)\}\}$

$\{implementation : \{/v/bren/pop.pas\}$

$pop\{real, real(*)\}\}$

$\{implementation : \{/v/bren/top.pas\}$

$top\{real, real(*)\}\}$

$\{implementation : \{/v/bren/empty.pas\}$

$empty\{boolean, real(*)\}\}\}$

If clients of stack want to use operations defined at deque then possibility relations need to be established.

$$\begin{array}{ll} create\ exec\_module : \{ & agent\ D : deque \\ & agent\ S : stack \\ & \quad possrel\ S\ push, pop, top, empty\ D\ push, pop, top, empty\} \end{array}$$

It may be that the S *push* interface may be OK at agent D, the S *pop* interface be OK at agent E, and the S *top* and S *empty* interfaces be OK at agent F. In this case we can use the modal operator ◇ to guide the compiler. The possibility operator ◇ indicates the existance of at least one world among D, E, F at which the interfaces of S are OK. The specification is given below.

$$\begin{array}{ll} create\ exec\_module : \{ & agent\ D\ E\ F : deque \\ & agent\ S : stack \\ & possrel\ S\ \diamond(push, pop, top, empty)\ D\ E\ F\} \end{array}$$

Example 3: Multiple Versions

Bertrand Meyer [Mey87] cites the example of table search to illustrate software that is “neither ever quite the same, nor ever quite another.” This means that while table search programs tend to do the same kinds of things, they are not exactly the same. A general description of the code would be:

1. Start at some position in the table *t*

2. Check to see if the search element exists at that position

3. If not, move to another position

4. Terminate the search either when the element has been found or the entire table has been searched

A more precise description is given in Figure 17. Details that may change (variants) include all the types and routines in uppercase. Now it is virtually impossible to write code corresponding to every combination of variant. However, it is likely that the few variants that are required will have already been written and exist in some library. These different versions of the table search constitute different worlds. Eg. there may be a search routine with an array implementation of table and one with a pointer implementation. A user who wants to change implementations from one to the other must consider both array_search and pointer_search as methods of one world. A change from array to pointer implementation is accomplished without alteration to any method.

If the array_search is being invoked from a main routine then the problem may be represented using the world interconnection language. The two implementations are made methods of the same world. The user selects the correct implementation

```
Search  (x:ELEMENT, t:TABLE_OF_ELEMENT) return boolean
        pos: POSITION
begin
        pos := INITIAL_POSITION(x,t);
        while not EXHAUSTED(pos,t) and then not FOUND(pos,x,t) do
        pos := NEXT(pos,x,t);
end;
```

Figure 17. A schema for table search

and establishes a possibility relation between the appropriate views.

$$\begin{aligned} \textit{world search} : \{\{ \;& \textit{implementation} : \{/v/bren/array\_search.pas\} \\ & \textit{a\_search} : \{x : int, t : int(+)\}\} \\ & \{\textit{implementation} : \{/v/bren/pointer\_search.pas\} \\ & \textit{p\_search} : \{x : int, t : pointer\}\}\} \end{aligned}$$

$$\begin{aligned} \textit{world main} : \{ \;& \textit{implementation} : \{/v/bren/example1.pas\} \\ & \textit{x\_search} : \{x : int, t : int(+)\}\} \end{aligned}$$

$$\begin{aligned} \textit{create exec\_module} : \{ \;& \textit{agent search} : \textit{search} \\ & \textit{agent main} : \textit{main} \\ & \textit{possrel main x\_search search a\_search}\} \end{aligned}$$

Alternately, array_search and pointer_search may be considered as separate worlds. In this case the representation becomes:

$$\begin{aligned}
&\textit{world array\_search} : \{ && \textit{implementation} : \{/v/bren/array\_search.pas\} \\
& && \textit{a\_search} : \{x : int, t : int(+)\}\} \\
&\textit{world pointer\_search} : \{ && \textit{implementation} : \{/v/bren/pointer\_search.pas\} \\
& && \textit{p\_search} : \{x : int, t : pointer\}\} \\
&\textit{world main} : \{ && \textit{implementation} : \{/v/bren/example1.pas\} \\
& && \textit{x\_search} : \{x : int, t : int(+)\}\} \\
&\textit{create exec\_module} : \{ && \textit{agent search} : \textit{search} \\
& && \textit{agent main} : \textit{main} \\
& && \textit{possrel main x\_search array\_search a\_search}\}
\end{aligned}$$

The three examples show different uses of the world interconnection model. A common thread is that the world interconnection model may be used to build extendible and reusable software.

CHAPTER V

SUMMARY, CONCLUSIONS, AND FUTURE WORK

Software productivity in large software systems may be improved by the reuse of multiparadigm software. This dissertation develops a possible worlds model applicable to the problem of software reuse. The model provides structure to a system composed of multiparadigm routines. This structure allows "used in" and "used instead of" relations between worlds based on the concept of views. A view consists of interface names and propositions. Each interface name has a corresponding interface pattern description and provides access to a method. Bindings between interfaces across multiparadigm worlds allows for multiparadigm software reuse. Propositions in a view may be used for verification, debugging or to guide the compiler.

A world interconnection language is developed based on the model. The description of the language is informal and intuitive. The language is used to illustrate multiparadigm software reuse.

Exposure of inheritance is a problem of some object-oriented languages. This model avoids this problem by developing possibility relations between agents based on views.

This work may be extended to include multiparadigm systems running on different architectures using different communication paradigms. Future work may include an implementation of the world interconnection system. This would include a system of compiler, linkers, debuggers, and interconnection language. The model may be extended to include the run–time. By keeping histories of runs, tools may be developed to reason about an agent's knowledge. The model holds the promise of a new programming paradigm based on the theory of modal logics and possible worlds.

This work could unify the current efforts towards a new generation of interconnection languages and systems.

BIBLIOGRAPHY

[Bal71] R. M. Balzer. Ports–a method for dynamic interprogram communication and job control. In *Spring Joint Computer Conference*, pages 485–489, 1971.

[BLP85] E. Balkovich, S. Lerman, and R. P. Parmelee. Computing in higher education: the athena experience. *Communications of the ACM*, 1214–1224, November 1985.

[BR87] T. Biggerstaff and C. Richter. Reusability framework, assessment, and directions. *IEEE Software*, 41–49, Mar 1987.

[Che80] B. F. Chellas. *Modal Logic.* Cambridge University Press, 1980.

[DGR81] Ph. Darondeau, P. Le Guernic, and M. Raynal. Types in a mixed language system. *BIT*, 21:246–254, 1981.

[EG84] B. Einarsson and W. M. Gentleman. Mixed language programming. *Software-Practice and Experience*, 383–395, April 1984.

[Ein85] B. Einarsson. *The Structure of Mixed Language Programming Realization.* Technical Report LITH-IDA-R-85-01, Linkoping University Sweden, 1985.

[Ein86] B. Einarsson. Mixed language programming realization and the provision of data types. *IEEE Software*, 2–9, 1986.

[Get84] J. Gettys. Project athena. In *USENIX Summer Conference Proceedings*, pages 72–77, June 1984.

[Gog86] J. A. Goguen. Reusing and interconnecting software components. *IEEE Computer*, 16–28, Feb 1986.

[Hai86a] B. Hailpern. Multiparadigm languages. *IEEE Software*, 6–9, Jan 1986.

[Hai86b] B. Hailpern. Multiparadigm research: a survey of nine projects. *IEEE Software*, 70–77, Jan 1986.

[Hal86] J. Y. Halpern. Reasoning about knowledge: an overview. In *Proceedings of the Conference on Theoretical Aspects of Reasoning About Knowledge*, pages 1–17, Morgan Kaufmann, 1986.

[HHS88] R. Hayes, N. C. Hutchinson, and R. D. Schlichting. *Integrating Emerald into a System for Mixed-Language Programming.* Technical Report 88–36, Dept. of C.S. The University of Arizona Tucson, Oct 1988.

[HMS88] R. Hayes, S. W. Manweiler, and R. D. Schlichting. A simple system for constructing distributed, mixed-language programs. *Software–Practice and Experience*, 641–660, July 1988.

[HN87] B. Hailpern and V. Nguyen. A model for object-based inheritance. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*, MIT Press, 1987.

[HS87] R. Hayes and R. D. Schlichting. Facilitating mixed language programming in distributed systems. *IEEE Transactions on Software Engineering*, 1254–1264, Dec 1987.

[JGM86] M. A. Jenkins, J. I. Glasgow, and C. D. McCrosky. Programming styles in nial. *IEEE Software*, 46–55, Jan 1986.

[KG87] R. R. Korfhage and N. E. Gibbs. *Principles of Data Structures and Algorithms with Pascal.* WM. C. Brown, Dubuque, Iowa, 1987.

[Kon86] K. Konyndyk. *Introductory Modal Logic.* Notre Dame Indiana, 1986.

[Kor86] H. F. Korth. Extending the scope of relational languages. *IEEE Software*, 19–28, Jan 1986.

[Kri63] S. Kripke. Semantical considerations of modal logic. *Zeitschrift fur Mathematische Logik und Grundlagen der Mathematik*, 9:67–96, 1963.

[LO90] T. G. Lewis and P. W. Oman. The challenge of software development. *IEEE Software*, 9–12, Nov 1990.

[LS77] E. J. Lemmon and D. Scott. *The Lemmon Notes.* American Philosophical Quarterly Monograph Series, 1977.

[Mey87] B. Meyer. Reusability: the case for object-oriented design. *IEEE Software*, 50–64, March 1987.

[MHS86a] S. W. Manweiler, R. Hayes, and R. D. Schlichting. *Adding New Languages to the MLP System.* Technical Report 86–9, Dept. of C.S. The University of Arizona Tucson, June 1986.

[MHS86b] S. W. Manweiler, R. Hayes, and R. D. Schlichting. *The MLP System Users Manual.* Technical Report 86–4, Dept. of C.S. The University of Arizona Tucson, Feb 1986.

[Mor79] T. More. The nested rectangular array as a model of data. *APL Quote Quad*, 9(4):55–73, 1979.

[Mor81] T. More. Notes on the diagrams, logic, and operations of array theory. In Bjorke and Franksen, editors, *Structures and Operations in Engineering and Management Systems*, Tapir Publisher Trondheim Norway, 1981.

[Not90] D. Notkin. Proxies: a software structure for accommodating heterogeneity. *Software-Practice and Experience*, 357–364, April 1990.

[Pri87] R. Prieto–Diaz. Classifying software for reusability. *IEEE Software*, 6–16, Jan 1987.

[Pur86] J. M. Purtilo. *A software interconnection technology to support specification of computational environments*. PhD thesis, University of Illinois at Urbana-Champaign, 1986.

[RGP86] C. V. Ramamoorthy, V. Garg, and A. Prakash. Programming in the large. *IEEE Transactions on Software Engineering*, 769–783, July 1986.

[SBK86] M. J. Stefik, D. G. Bobrow, and K. M. Kahn. Integrating access-oriented programming into a multiparadigm environment. *IEEE Software*, 10–18, Jan 1986.

[Sea87] IEEE Software. *Seamless Systems*, Nov 1987.

[Sny87] A. Snyder. Inheritance and the development of encapsulated software components. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 219–252, MIT Press, Cambridge, MA, 1987.

[TP86] H. Tu and A. J. Perlis. Fac: a functional apl language. *IEEE Software*, 36–45, Jan 1986.

[WSHF81] W. Wulf, M. Shaw, P. Hilfinger, and L. Flon. *Fundamental Structures of Computer Science*. Addison-Wesley, 1981.

[Zav89] P. Zave. A compositional approach to multiparadigm programming. *IEEE Software*, 15–25, Sept 1989.

VITA

Brendan Machado

Candidate for the Degree of

Doctor of Philosophy

Thesis: A MODEL FOR SOFTWARE REUSE IN A MULTIPARADIGM ENVIRONMENT

Major Field: Computer Science

Biographical:

Personal Data: Born in Bombay, India, March 22, 1963, the son of Raphael and Grace Machado.

Education: Attended St. Stanislaus Junior College, Bombay, upto May 1981; attended St. Xavier's College, Bombay, upto May 1985; received Bachelor of Science Degree in Physics and Chemistry from The University of Bombay in May 1985; completed requirements for the Doctor of Philosophy degree at Oklahoma State University in December, 1991.

Professional Experience: Teaching Assistant, Department of Computer Science, Oklahoma State University, August, 1987, to May, 1991.