INFORMATION TO USERS

The most advanced technology has been used to photograph and reproduce this manuscript from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book. These are also available as one exposure on a standard 35mm slide or as a 17" x 23" black and white photographic print for an additional charge.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

U·M·I

University Microfilms International
A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
313/761-4700 800/521-0600

Order Number 9005992

Typing mechanisms in software engineering environments

Kwon, In Sup, Ph.D.

Arizona State University, 1989

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

TYPING MECHANISMS IN SOFTWARE ENGINEERING ENVIRONMENTS

by

In Sup Kwon

A Dissertation Presented in Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

ARIZONA STATE UNIVERSITY

December 1989

TYPING MECHANISMS IN

SOFTWARE ENGINEERING ENVIRONMENTS

by

In Sup Kwon

has been approved

August 1989

APPROVED:

Timothy E. Lindquist, Chairperson

Robert J. Fornaro

James S. Collofello

Terry Mellon

F. Golshani

Supervisory Committee

ACCEPTED:

R E Bowles

Department Chairperson

Neil Hadley

Dean, Graduate College

ABSTRACT

In the emerging concept of the software engineering environment, serious problems with the integration of software tools and data in the software development lifecycle have been revealed. The more automated software tools are required to assist the software development, the more complex integration on the tool data.

There are currently several ongoing research efforts attempting to solve these problems. Despite many research topics available, virtually no research deals with the typing of the tool data to aid integration of various software tools. Several existing typing theories try to formalize the various kinds of data manipulated in a programming language. However, applying a typing mechanism to software engineering environment data allows more detailed description of information communicated among tools.

This research presents the problems in building the software engineering environment, its software architecture, and typing mechanism for software tool data. The entity-relation model together with an object-oriented paradigm is used to design the software tools. Analysis of the proposed system reveals the design quality and productivity improvements for software development. The explicit polymorphic mechanism is used as an underlying formalism. The typed software engineering environment system is a self-evolving system encompassing the creation and management of types for tool data in software engineering environments.

TABLE OF CONTENTS

1. INTRODUCTION

There has been significant development recently in computer technology, both in hardware and software. More sophisticated and powerful systems are now available for a wide range of business and engineering applications. Software engineering serves a major role in developing such systems by providing the basis for a good scientific methodology from requirements engineering through maintenance of software systems.

1.1 Problems in Software Development

Over the past 20 years, software developers have built tools for software development as a way to support better production of software systems. However, the practices in software development faced serious problems with demands for increased functionality, reliability, and user friendliness of software systems. The software systems to be developed to fulfill such demands require more complex structures. The typical problems can be summarized as:

1) the object size of the software system is rapidly increasing,
2) the complexity and increased object size of software system requires large-scale development activities,
3) the resulting software does not meet the requirements or delivery deadline,
4) the resulting software is neither reliable nor efficient,
5) the documentation is poor, and
6) maintenance costs of delivered systems are ever increasing.

To overcome such problems, new paradigms for software development are introduced. These paradigms include approaches such as JSD (Jackson System Development), software requirement engineering, specification

technology, modularity, information hiding, project management, cost estimation, and so on.

The other important trend is that more software developers are now dependent on automated tools for project management, program generation, prototyping, or report generation [Rich and Waters 1988]. In the presence of a wide variety of tools, the problem of integrating those tools still remains difficult. Heterogeneous tools integrated in a single software environment usually result in a less reliable environment. Also, most of the currently available tools only support a narrow range of the software life-cycle.

When a number of specialized tools run together, inter-application communications are usually processed by operating system level services with a limited set of predefined data types such as files. However, this method does not guarantee an efficient management of highly specialized, complicated software engineering environment data, which are becoming more and more abundant. It further ignores relationships between data.

The concept of a software engineering environment emerged in the mid-1970s to provide more cost-effective and rapid development of reliable software systems. The initial approach with software engineering environments was to integrate those available software engineering techniques and experiences with appropriate methods and tools in a synergistic manner. The common purpose of building an environment is to support the efforts of individuals, groups, and project teams over the entire software lifecycle. The initial attempts at developing software engineering environments involved modern operating systems where tools are invoked as programs and data are organized as files such as in Unix and VMS . Other approaches include syntax-directed environments such as the Cornell

Program Synthesizer [Teitelbaum et al. 1981] and Gandalf [Habermann and Notkin 1986], language-based environments such as Smalltalk [Goldberg 1980; Goldberg and Robson 1983], and object-oriented database management systems [Kim et al. 1987; Lamsweerde et al. 1988; Penney and Stein 1987].

In defining good interfaces among tools supporting reusability, adaptability, and necessary abstractions in software development, a more efficient mechanism is needed to manage various types of tool interfaces including data, programs, and even processes, with the aid of a type management facility and appropriate compiler. Fundamental research is needed for reliable and cost-effective methods to support well-connected environment systems, and a well-defined typing mechanism for tool data which will support dynamic creation as well as evolution of types.

1.2 Research Objectives and Approach

In a software engineering environment, software tools are used to develop software systems. The software tools use tool data to share information among themselves. The tool data can be in many forms such as the *basic types* of integer, real, and character, the *structured types* of record, array, and file, and the *process types* to describe the running image of program. The tool data are the objects and relationships among those objects that tools share. To support independent management of such tool data, it is necessary to make the tool data appear as autonomous objects managed by the software engineering environment.

There has been much research to support smooth interfaces among tools in a software engineering environment, yet research concerning typing mechanisms for tool data is still new. The objectives the research herein reported are to model a well-defined typing mechanism to efficiently manage

tool data in a software engineering environment, to analyze the design feasibility of such a type system, and to assess potential benefits from such a system. The primary focus of this research deals with aspects of tool data management for the efficient integration of tools in software development, and the integration of application programs running in the software development process using tool data for inter-application communications.

The research into a typing mechanism for software engineering environments includes the design of an environment type processing language to define environment types and manipulate those types, design of the typing system for a software engineering environment, and analysis of such a typing system. This environment type processing language uses explicit polymorphic types to support the various requirements of tool data. The design analysis of the typing system consists of use analysis for the type system user and implementation analysis of the typing system in the software engineering environment. The use analysis identifies the design guidelines used in software engineering environments when tool builders use the tool data and the tools for integration. The implementation analysis focuses on the issues of feasibility of implementation of the typing system and the issue of compatibility of the typed system with host languages such as Ada in the CAIS (Common Ada Programming Supporting Environment Interface Set) environment. The expected quality and productivity improvement using this typed software engineering environment is finally examined.

1.3 Organization of Dissertation

The dissertation begins with the introduction of general software engineering environment concepts in Chapter 2. Chapter 2 includes the definition and classification of software engineering environments, other related research

on this subject, the general conceptual architecture of a software engineering environment, and required characteristics of a software engineering environment.

The object-oriented development paradigm is discussed in Chapter 3 to provide related background information on the research. The object-oriented features of tool data are discussed in this chapter.

Chapter 4 describes the environment type processing language that defines and manipulates the environment types. Environment types, tool data, and the type constructor are discussed. Related materials about CAIS features are presented later in Chapter 4.

Chapter 5 describes the design of the environment type management system in detail. All the component systems of the environment type management system and their functions are described.

The analysis of the typed software engineering environment in Chapter 6 includes use analysis for the environment type user's view, implementation analysis for the type system builder's view, and analysis of expected quality and productivity improvement using this typed software engineering environment.

The conclusion in Chapter 7 summarizes the presentation of the typed software engineering environment. It concludes by identifying the contributions of the research, and discussing the remaining future work.

2. THE SOFTWARE ENGINEERING ENVIRONMENT

An environment is a system consisting of hardware and software tools with a set of interfaces among the tools to manipulate information. System developers use this information to build software systems. The software engineering environment includes the programming environment that supports a coding effort as well as all activities in programming-in-the-large. The specific purpose of the environment can be different from one user to another, although the general goal of the environment is to enhance the development productivity and the quality of any resulting software system. With a powerful software engineering environment, the system developer can increase the productivity to build the software system using software tools and appropriate methodologies.

2.1 Definition of a Software Engineering Environment

The phrase "software engineering environment" has been used in many ways with no consistent definition, probably because the specific purposes and roles of a software engineering environment were not revealed clearly until the recent development and use of software tools. However, the term "programming support environment", which is too frequently used interchangeably with software development environment, is defined in the IEEE Standard Glossary of Software Engineering Terminology [IEEE 1983]. According to the IEEE definition, the programming support environment is an integrated collection of tools that provide programming support capabilities using simple command languages. A useful definition of the software engineering environment can be derived by taking related definitions. If an environment is defined as a collection of tools used to build a software system [Dart et al. 1987], then a software programming

environment is a collection of programming tools the developer uses to build software systems.

The purpose of the software development environment is mainly to automate and augment activities in software development such as programming, design, project management, etc. To support all the activities and scopes of software development, a supporting environment must include not only tools but also methods that are currently being used in software development (state-of-practice).

Since the software engineering environment is an integrated system with several heterogeneous tools, the component software tools and methodologies must be managed by an automated mechanism to create, maintain, and access the component software tools by a well-defined, underlying management system. In this respect, a software engineering environment can be defined as an integrated system to manage a collection of software tools and methodologies to provide automated assistance for software developers to build objective software systems.

The software engineering environment should provide support for the activities in program development as well as for those in programming-in-the-large, which covers all activities of the software development lifecycle. Among many research issues regarding software engineering environment, this research focuses on the behavior and the structure of the software tools and tool data as well as the management system for them. It is necessary to provide a well-defined mechanism for the software engineering environment to manage the tool data in a more consistent and controlled manner if the software engineering environment is truly a system to automate the software project.

2.2 Classifying a Software Engineering Environment

There are several known software engineering environments either in practice or in research. We categorize them in terms of the underlying system paradigm as follows: operating system shell, language-based environment, syntax-oriented environment, toolkit environment, and database-oriented environment.

2.2.1 Operating System Shell

Modern operating systems such as Unix and VMS provide various facilities in their shell to automate the simple, yet time-consuming, activities in program development. Unix automates tool support to some degree in project management. The pipe and redirection of standard input and output manage data among tools. The shell command language simplifies many unnecessary commands from the user interface. VMS provides a unified format for object codes among different languages in the program development. VMS supports a version control mechanism for files managed in development activities.

The architectural view of an operating system shell environment reveals that it is most primitive in the area of tool data management and integration. The tool is a simple program in the form of an executable image. The tool data is simply a file that can be redirected to another tool. The control of those tools or tool data is executed by command languages. In such an environment, there is no well-defined method to manage the tools or tool data. The user must control tools to manipulate the corresponding tool data correctly according to their implicit rules. The types of tool data are not provided such that every tool data has the same type, i.e. basic type of files [Kernighan and Mashey 1981].

2.2.2 Language-Based Environment

The language-based environment provides a set of tools that are built for one particular language. The tools in such an environment are highly interactive and offer some degree of programming-in-the-large facilities. Every component in the language-based environment is specific to the objective language for which the environment is built. Examples for such environments are Rational Environment [Archer and Devlin 1986] for Ada development, Interlisp for LISP language [Narayanaswamy 1988; Steele 1983], DIANA-based environment for Ada [Rosenblum 1985], Cedar for Mesa/Cedar [Swinehart et al. 1985], Smalltalk for Smalltalk [Goldberg 1980], and others.

Since the entire environment is to support the development and execution of programs written in a single language, the user has very few options for non-programming purposes. However, such an approach makes tool integration very consistent, and all resources are accessible directly from the programming activity. Also, the user can easily learn and run the machine even though the system has limited resources. The main disadvantage of such an environment is that it supports few services other than writing code and executing it.

Currently, some advanced language-based environments support various tools around the programming environment and attempt to enrich functionality. Incremental development of coding, debugging utilities using a common form of object code, version control, and syntax-directed editors are examples of such environments. Such tool sets are well-suited for programming-in-the-large only when the whole project is written in one language [Archer and Devlin 1986].

2.2.3 Syntax-Oriented Environment

To eliminate simple yet time-consuming tasks of the Compile-Link-Go paradigm, the syntax-oriented environment supports an intelligent editor that checks the syntax of the program with the rules of the underlying programming language. The environment provides the tools for pattern matching, debugging, attribute grammar, and so on. The Cornell Program Synthesizer (CPS) [Teitelbaum et al. 1981] and Gandalf [Habermann and Notkin 1986] are examples of this environment category.

The motivation to develop a syntax-oriented environment was to provide the user with interactive tools to access the underlying program structure. Starting with the syntax-directed editor, the environment supports programming-in-the-small. The user can construct a program using a template to synthesize a fully functional program. The environment manages the underlying structures for program syntax and semantic information, and provides graphical representation of the structure such as in Pecan [Reiss 1985].

Such an environment has the capability to generate instances of a syntax-oriented environment automatically. This feature makes it possible to support a multi-language environment with little development effort. The environment builder can specialize a project-specific language environment and restrict the syntax checking. The syntax-oriented environment's main focus is on coding activities, and it is not suitable for all phases or activities in the lifecycle. The industry has been reluctant to accept such an environment in spite of some popularity in academia. Since the syntax-oriented environment is built on the homogeneous structure for programs, adapting

different tools with foreign structures into the environment is not a trivial task.

2.2.4 Toolkit Environment

The toolkit environment consists of a collection of tools to support the language-independent programming effort in programming-in-the-large. It starts with the operating system and adds programming tools such as compilers, linkers, debuggers, editors, and assemblers as well as large-scale development tools like version control, configuration management, and project management. The toolkit environment has a data modeling capability to provide the extensibility and portability. Examples are Unix Programmer's Workbench (Unix/PWB) [Kernighan and Mashey 1981], Macintosh Programmer's Workshop (MPW) [Meyers and Parrish 1988], DEC VMS VAX-set [DEC 1984], Portable Common Tool Environment (PCTE) [ESPRIT 1986], and CAIS [CAIS 1986].

The Macintosh operating system provides a customizable and extendable environment because it is constructed with several resources. Such resources for both operating system and applications include data segment, code segment, device driver, window object, bit-map image for graphical display, icon, sharable memory-resident data, etc. On top of such an operating system, the MPW environment provides an extendable set of tools for compiler, linker, editor, and source level symbolic debugger. All the object formats for different languages have identical structures to provide consistent integration and user interface. The resources in the Macintosh operating system are predefined and not user definable as new resource type. The structure of resources is managed by the operating system without semantic information. The Unix operating system is similar in its extensibility and

limited degree of portability. Unix provides a simple data model as a byte stream to support both file and persistent data. The Unix environment can be enhanced with new or modified tools without much difficulty.

In such environments, the underlying tool data structure is simple and has no support for semantic information that the environment system can control. This results in a serious limitation of tools for incremental development capability. Since the environment can process only predefined types for tool data, there is a potential problem for managing tool data in incompatible tools.

The PCTE and CAIS tool substrates a persistent object and some degree of typing of objects by extendable attributes. However, they do not support well-defined type-checking facilities in a uniform and automated manner. The details of those environments are discussed in the sections 2.3.4 and 2.3.5.

The toolkit environment evolved from an operating system based environment to satisfactorily integrate tools with language independency. It enhanced development capability by version control tools such as VMS SCCS (Source Code Control System) and CMS (Code Management System). The toolkit environment essentially integrates tools to support large-scale project development with multi-language and multi-tool management for uniform user control. Still, it lacks user definable management of tool data over tools. The simple tool integration characteristics of such environments cannot achieve an appropriate maintenance mechanism for a large software system.

2.2.5 Database-Oriented Environment

This approach was created to handle project management related data in development phases. The database-oriented environment allows a current database management system to store and share management data such as in

configuration management and project management among project teams. Typically such a system does not directly support tool-related data management. Other approaches include some degree of extension for tool data management by employing a language-oriented or toolkit environment. One example of a database-oriented environment is Gemstone [Penney and Stein 1987].

Often the database-oriented environment is based on an object-oriented database management system that manages software engineering environment information in a central repository with an object-oriented approach and functional query languages. One of the motivations for these approaches is that the conventional database management system is not suitable for software engineering environment data. In a software engineering environment, the number of data is small while the variety of data is large and specialized, in contrast, the conventional database management system deals with large number of data but small variety. The object-oriented database approaches also pursue rapid prototyping, fast implementation of objective software, easy maintenance, and easy modification [Kim et al. 1987; Lamsweerde et al. 1988; Penney and Stein 1987]. However, although it is a useful concept for software engineering environments, it does not support in significant measure a specific methodology for tool development. One of the limitations is that the schema for tool data definition is static to the environment database, and the user cannot create a new type of tool data without "remodeling" the schema.

2.3 Related Works

There are numerous approaches to solving design problems for a software engineering environment. Many of them are still project specific and do not

support general criteria for a tool data management system. The following research is most closely related to this research.

2.3.1 Arcadia

The Arcadia Consortium focuses on developing architectural principles for the creation of software development tools [Taylor et al. 1986; Taylor and Standish 1985]. Initial attempts were made to build a series of prototyping tools for Ada programs under the architectural principles for software tools. The Arcadia system is currently under development.

Arcadia has type-based objects and stores them in repositories. All objects are typed and are instances of some abstract data type. The objects can be persistent and manipulated by tools. Examples of the objects are source code, object code, symbol tables, test data, test results, bit-mapped display frames, and text. Object type is also an object as an instance of type. Arcadia supports relationships for the tools and the objects.

The strong typing mechanism for the objects prevents an incorrect use of objects by the tools. Many of the essential tools are built into the repositories as tool fragments to construct the customized tools. Tools can be activated by procedure calls in the tool code with static or dynamic binding.

2.3.2 Gandalf

Gandalf is a syntax-directed environment that employs the Unix file system as a mechanism for repositories [Habermann and Notkin 1986]. It provides an advanced structure for syntactic information of the program by a graph structure in a symbol table. The user shares common design criteria for program development by using the underlying structure. Data persistency is maintained by the file system for long-term development while several variations of the syntax tree reside in process memory. Access control to the

tree is maintained by a Unix protection scheme. Programming language can access and manipulate the tree. The tools communicate with each other by sharing the same tree in a common repository. All the tool data (objects) are represented in tree form and named as Unix files. The portability of tools is supported by Unix.

2.3.3 Gaia Project

This approach intends to support the full characteristics of the object-oriented paradigm for Ada programming. Gaia is a framework to provide a common set of machine-independent interfaces supporting integration of tools without modification. It provides portability, X window-based user interface, environment definitions by inheritance, class management, extensibility, an object-oriented mechanism, and a prototyping mechanism. However, it has no support for explicit and flexible attribute handling, and it is not complete in class definitions for attributes in a user-definable manner. Since it is based solely on an object-based model, it lacks support for the large number and complex structure of relationships among components of a system [Vines and King 1988].

2.3.4 PCTE

PCTE is a set of interfaces being built for the development of Ada software running in the Unix environment. It serves as the interface to manage host-system accesses for tool builders. Therefore, it is a Unix-oriented common interface for both C and Ada. It is now under development by contract with the European Strategic Programme for Research and Development in Information Technology (ESPRIT) of the Commission of European Community (CEC). It is designed according to the object management system (OMS) using a database management system. Although the PCTE work

shares many of the goals of the CAIS effort, it lacks the security requirements and user-definable typing mechanism for tool data in the complex and rapidly growing kernel environment, which usually deals with a large number of relationships among objects [ESPRIT 1986].

2.3.5 CAIS

CAIS is a set of operating system level interfaces to support source-level transportability of tools among APSEs (Ada Programming Supporting Environments). The development of CAIS was initiated by the DoD (U.S. Department of Defense) Ada Joint Program Office (AJPO), designed by the KAPSE (Kernel APSE) Interface Team (KIT) [Oberndorf 1985], and its operational definition is now running under Sun Unix and VAX VMS. CAIS serves as a set of interfaces to access the KAPSE layer; these interfaces translate the service calls to host system dependent code. CAIS provides general facilities of a software engineering environment including file management and process management, and is now under revision to provide a typing mechanism for the CAIS node model as well as the transaction and triggering mechanisms.

CAIS is based on the entity-relationship-attribute model, which uniformly supports data abstraction with a simple interface. CAIS interfaces are designed as Ada procedures and functions in the set of Ada packages. CAIS is an open-ended system and can be self-evolved over time and on different host systems. The entity management system (EMS) of CAIS is based on STONEMAN requirements [Buxton 1980] using the entity-relationship concept. CAIS is designed to cover a broad range of tools and classes of projects [CAIS 1986].

The Entity Management System (EMS) of CAIS includes entity, relationship, and attribute. The entity is the representation of real world objects. The relationship is a directed N-ary association among entities. The attribute associates entity or relationship with relevant values. The typing requirement of CAIS in EMS is to organize entities, relationships, and attributes as a partition over sets of types. It is capable of specifying entity type, relationship type, and attribute type. The relationship type supports both functional mapping (1 to 1 or many to 1) and relational mapping (1 to many or many to many). The requirement for typing in CAIS provides the mechanism to relate data, relationships, and properties of data or relationships. The security constraint is processed by access control to check the validity of operations over data. The typing of CAIS supports inheritance of attributes, relationships, and operations. The requirements for the typings of CAIS specify to have type definition, type change, triggering mechanism to be invoked whenever a change occurs [Oberndorf 1985].

2.4 General Architecture of a Software Engineering Environment

The services that a software engineering environment provides vary depending on the scope of the environment. The scope of a software engineering environment can be generally classified in three aspects: project purpose, developer role, and lifecycle.

The scope of **project purpose** describes specific characteristics of each project to develop certain end-application software. A software engineering environment can be classified with scope of project purpose, for instance, as a database generation environment, a real-time simulation environment, or a knowledge-based expert system generator shell.

The scope of **developer role** describes the specific user roles of the software engineering environment such as project manager, programmer, designer, configuration management personnel, etc.

The scope of **lifecycle** describes the time-span covered by the software engineering environment. Software engineering support can be provided for earlier phases of the software development lifecycle, whereas most of the programming environment focuses on later phases of the lifecycle scope.

It is important to understand the roles of a software engineering environment to ascertain its function in different situations and for different purposes. Using these aspects of scope, we categorize a general class of developer roles.

2.4.1 Roles in Software Development

In modern software engineering practice, the specialization of software production has created many roles for the software developer. To focus on the functions of each user to create, manage and use a software engineering environment, we can categorize the developer roles of software development as environment builder, environment adapter, and project user. Figure 2.1 shows the relationship among these roles in an integrated environment.

The **environment builder** establishes the basic and general-purpose tools and defines general-purpose (or generic) types of tool data. This role allows the environment components to grow in a dynamic way while the environment is being used. The environment builder usually constructs the types of tools or tool data by using a tool that defines or creates types and objects. This tool is one of the basic building blocks for a general environment component. The special tool to provide such an environment component consists of two special tools: an Environment Type Definition Processor

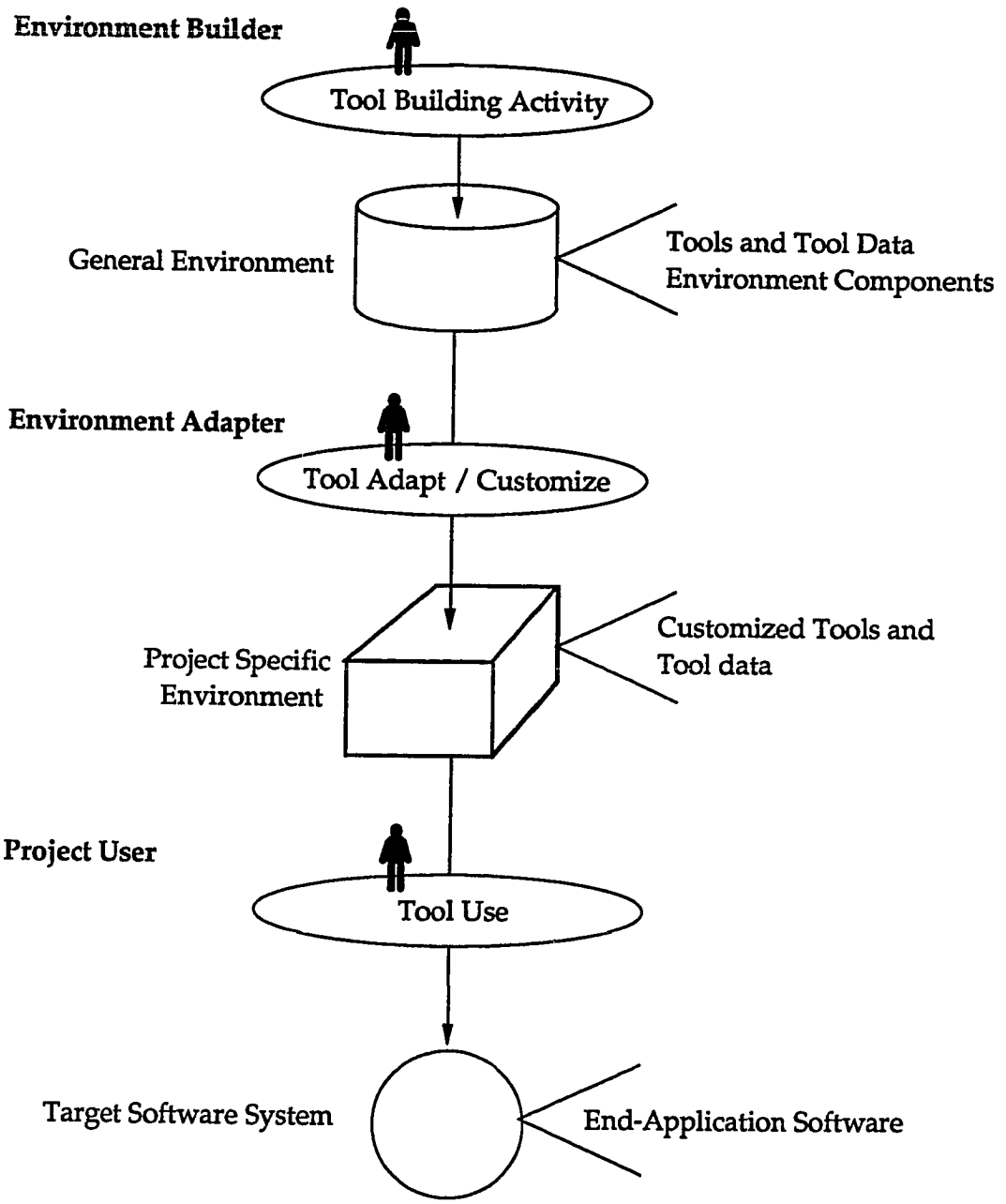


Figure 2.1. Roles in Software Engineering Environment

(ETDP) and a Tool Compiler. We shall examine the details of these tools in a later chapter.

The **environment adapter** adapts general-purpose tools or tool data to project-specific tools or tool data. This process is usually done by customizing the general-purpose tools or tool data to the project user's requirements. The adaptation will be done in the desired scope of the environment; for example, if the project user is using the resulting environment for program development, then the environment adapter as project manager can furnish more specialized tools for program development activities. The scope of project purpose, developer roles, and lifecycle should be considered for adaptation of tools and tool data.

The **project user** develops the objective software system using the project-specific environment. The project team member shares a common interface among various tool data on the project-specific environment. The team member can create new types of tool data if the use of such tool data among team members is frequent enough to create new types. However, creating new types of tools or tool data is an environment adaptation activity that the environment adapter performs as well.

The roles in a software engineering environment can be further specified for specific functions of each project team member and manager. This categorization is not by personnel but by functions of the development activity.

To summarize, a software development activity includes various roles for the involved members. It is necessary to differentiate the functions of members both to maintain the software engineering environment and to manage tools and tool data. This categorization of roles is especially

important because the software engineering environment is to be self-evolving throughout the software lifecycle.

In the next section, the roles of project developers in the software engineering environment are described from an architectural viewpoint.

2.4.2 Multi-Layered Architecture and Integration

A software engineering environment is composed of many heterogeneous components. It has a hierarchical structure of services among tools in several layers to support various services from different sources of requests. A software engineering environment can be built on top of a virtual operating system which, in turn, is built on top of native operating system such as Unix or VMS and provide portable interface supports to the structure of higher layer. Tools are built in a software engineering environment.

The scope of a language oriented environment cannot be expanded to support a broad range of interface services in a vertical or horizontal structure. As an example of a language-oriented environment, the Ada programming environment can provide rich functionality for building the stand-alone Ada software system. However, it does not support a very flexible mechanism to integrate other Ada programs in its language-oriented environment (horizontal integration).

The operating system supports the interface between the software system and hardware to provide appropriate hardware services (vertical integration). There is an important interface missing among software systems, which is the interface among tools in the software development phase. This missing interface will be filled by the typing mechanism presented in this dissertation.

The software engineering environment must support tool-building services as well as integration-management services among tools. The tool-building services include reusable type definitions via an inheritance mechanism and their appropriate management. The integration-management services include customizing and tracing of functionality of tools for tool builders' requirements. When tool builders can define their own types of tool data and tailor them to meet their purpose, the software engineering environment will provide self-evolving services in the dynamic situations of software system development.

The environment user who has the various roles of project management, tool construction, or tool use for objective software development interfaces with the environment via command language. The carriers of information between the environment user and the environment can be specified as environment data. The environment data can be further specified in terms of where it interfaces. The environment data can interface between the environment and tools, or among tools.

The environment itself consists of several layers, from hardware to environment user. The operating system controls hardware and provides services such as command language interpreter or application software to the requesting system. If the developer uses a set of tools (operating system tools) such as high-level language to develop objective software system, then the user will interface with the tools to get the services of the operating system which, in turn, requests services to the hardware. The command shell (overlaying environment) interfaces with the tools instead of calling directly to the operating system services. The user will create and manage the tools via the command shell if it is rich enough to support all required

characteristics of the software engineering environment. Figure 2.2 shows an architectural view of the conventional operating system based environment. In this architecture, the environment is simply all the layers from operating system to tools.

The specific roles of a software engineering environment are all included in a single tool set. Each tool can either directly call the services to operating system or use an overlaying environment such as a shell.

In this software engineering environment, many interfaces among several underlying layers result in many integration methods among tools. For example, a text editor can be coded either by using full implementation of machine level instruction or by customizing existing tools provided by the operating system. The interface method between two implementations can be defined in ad hoc specifications for specific requirements. The future text editor that intends to integrate such tools should be implemented either with full design and implementation knowledge of existing tools or with specific interface rules for existing tools. The resulting design of tools in such an environment enforces only tight integration among tools. This results in difficulties in the software project with a large number of teams and a long period of development cannot afford. It is an especially serious problem in programming-in-the-large. Figure 2.3 shows how the architecture for the software engineering environment separates environment user roles according to the use or management of a software engineering environment. This architecture supports several interfaces among the layers from hardware to user interface to the environment.

The **operating system layer** provides services from the hardware. The implementation of this layer is machine dependent. However, the portability

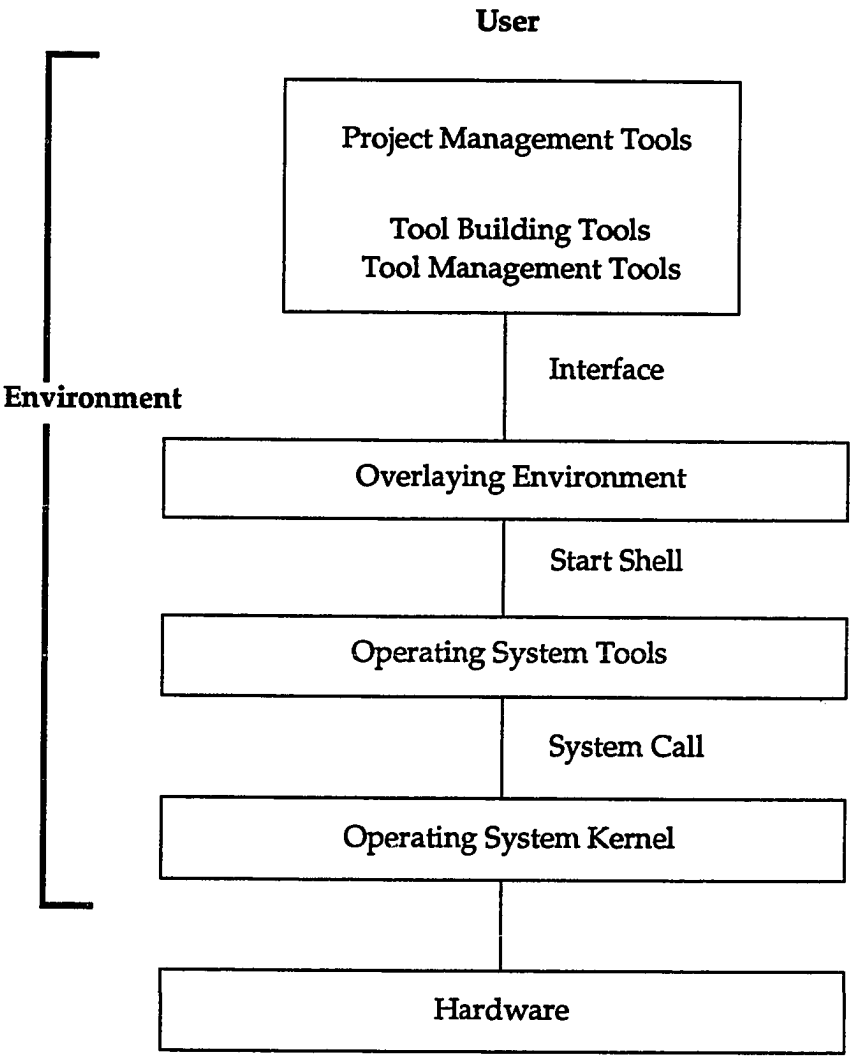


Figure 2.2. Conventional Architecture for Software Development

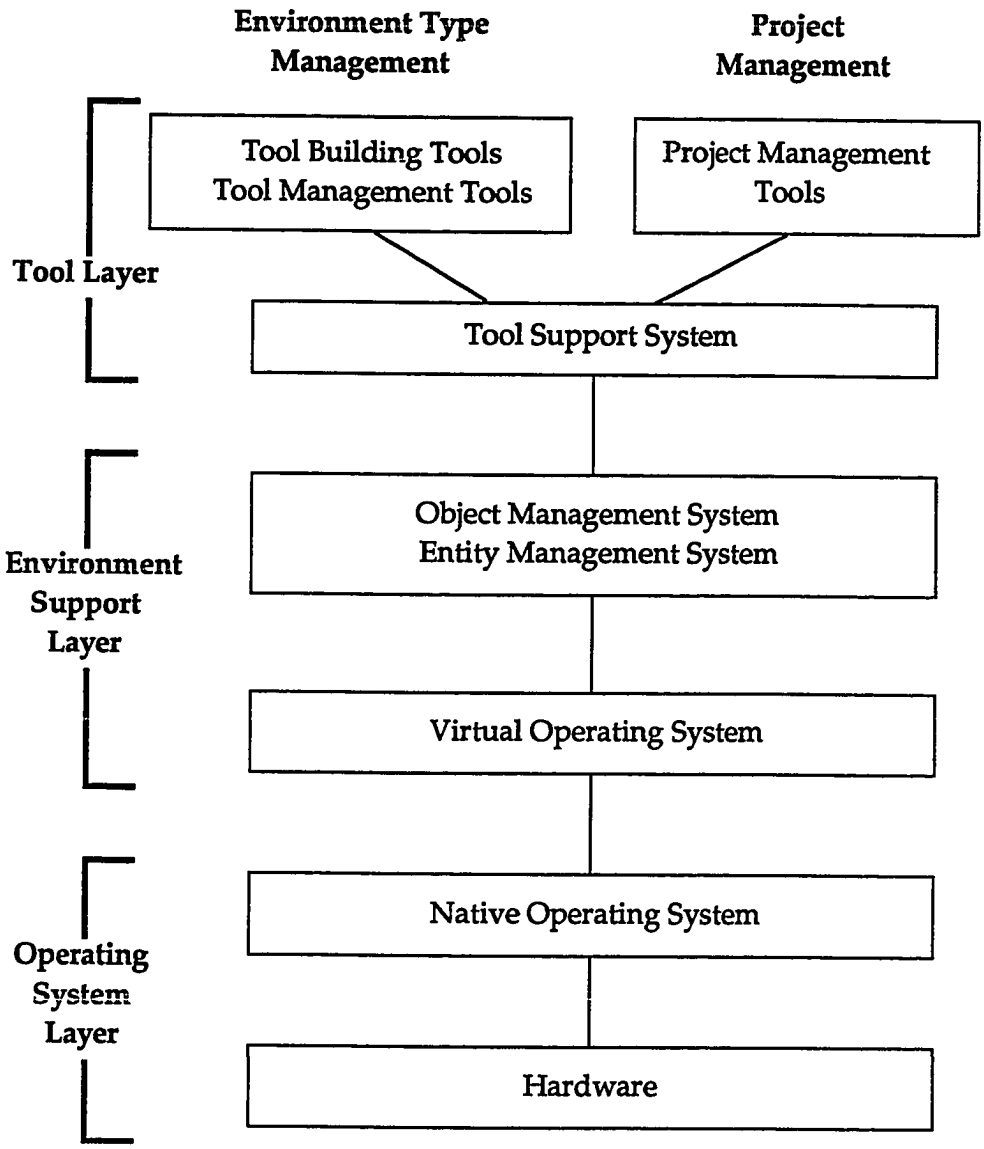


Figure 2.3. Conceptual Architecture for Software Engineering Environment

[Source : Penedo and Riddle 1988]

of this layer can be partially supported if the implementation of the native operating system uses a commonly supported host language among different machines. The purpose of the operating system layer is to access and control the hardware. The interface to the operating system layer is specified as a set of system calls to the native operating system.

The **environment support layer** provides the services of environment component management. The environment component in this layer can be the object in an object management system or the entity in an entity management system. The environment support layer interfaces with the operating system layer to manage the environment component. This layer can be fully portable between different machines and operating systems.

The virtual operating system provides a set of general operating system calls that are universal for all native operating systems. The purpose of the virtual operating system is to make service interfaces from the higher level transparent and the environment portable to a wide variety of hardwares and native operating systems. The interface specification of the virtual operating system must provide a rich and complete set of parameters to select a specific native operating system call.

The object management system (OMS) provides an object-oriented model to define, organize, access, and instantiate the object of the software engineering environment. The object of the environment is an abstract data type as a *carrier of information* to supply the data to tools (simple or constructed type of data) or to control the data (process). In a later chapter we will discuss the details of the object-oriented model. The entity management system (EMS) creates, organizes, and accesses the entity and relationship of the entity-relationship model. In OMS/EMS, the lifecycle of the object or

entity as a carrier of information is managed. The higher level system simply requests the services to OMS/EMS in consistent interfaces. The OMS/EMS interfaces with the virtual operating system to allocate and access computing resources such as memory or file to construct and manage the objects of the environment.

The **tool layer** provides services to manage tools and tool data. Each tool and tool data consists of single or multiple simple objects. The construction of tools, the definition of tool data, and their management are done in a consistent and unifying manner. The environment user interfaces with the software engineering environment through either a call to tools or the command shell, which is a simple tool to interpret the user request to call tools. The tool layer provides two specific roles for environment management and project management.

The roles for environment management deal with the management of tool and tool data for the environment user. The tool support system manages tool data. The environment type for tool data is defined and cataloged in the environment type library as a reusable component of software development by an environment type definition mechanism. The tool support system integrates the tool data for high-level, role-specific tools. The access mechanism to the tool data is managed at this time.

The environment type management activities of the environment builder or the environment adapter uses the tool-building tool or tool-management tool. The project management roles for software development uses the project management tool to access and use the tools. These activities include roles from programmer to project manager.

The integration of several layers as well as tools and tool data should be supported in a software engineering environment. Interfaces between user and environment and among tools must be well defined. The user interface must be easy and familiar to the existing practice of software development. The current trend of user interface moves from command interface to menu driven or graphical interface [Meyers and Parrish 1988]. The database interface provides a repository of tool data in the library, file system, or database management system [Burton et al. 1987]. The implementation of a database interface is environment-specific so that the efficiency of tool data is maximized while the user interface to such tool data is consistent and transparent.

2.5 Required Characteristics of a Software Engineering Environment

A software engineering environment should support the following features:

1) The methodology and automation for all phases of the software development lifecycle, and
2) The general tool to provide most commonly used functions such as editor, compiler, and library management [Houghton 1987].

The required characteristics can be further specified as the aspects of utility, usability, adaptability, degree of automation, degree of integration, and cost-effectiveness [Penedo and Riddle 1988]. The *utility aspect* requires

1) coverage of user roles such as programmer, designer, project manager, quality assurance personnel, and tester, and
2) coverage of lifecycle activities from requirement engineering to maintenance.

The *usability aspect* requires

1) user friendliness,

2) easy and consistent user interface of tools, and

3) closeness of tool functions to existing practice.

The *adaptability aspect* requires the ability to customize project-specific characteristics as well as an individual developer's preferences.

The *degree of automation* should support the automated assistance for time-consuming, routine jobs in software development.

The *degree of integration* should support

1) a consistent user interface among relevant tools and

2) code and/or data sharing.

The *cost-effectiveness aspect* requires that the software engineering environment meets long-run economics in software development; that is, the use of the software engineering environment must enhance the productivity of software development and the quality of the resulting software system.

Although no existing software engineering environment supports all the characteristics outlined above, such an environment should furnish the adaptability, automation, integration, and cost-effectiveness. To provide those characteristics, we can summarize the minimally required features for a software engineering environment:

1) it must support a broad range of user roles that includes programmer, designer, and maintenance team as well as the entire lifecycle activities;

2) it must be analyzable;

3) it must be customizable to a specific purpose;

4) it must be extendable (incrementally definable);

5) it must be traceable; and

6) it must support integration.

These desirable characteristics cannot be achieved easily without well-designed underlying structures. With a vast amount of information and functionality required, software engineering environments will fulfill the desired tasks when the *a priori* design of structure and behavior is sound and complete. This research is based on these required characteristics for a software engineering environment and provides an essential mechanism to fulfill these criteria.

2.6 Dynamic and Static Features of Software Tool Data

In a software engineering environment, the types of tool data can be predefined (system supplied) and user defined. The standard input or output is an example of predefined types for files. To provide a rich set of tool data and to manipulate them in the most productive way, it is necessary to support user-definable types for tool data. The software engineering environment can provide a set of predefined tool data types when it is installed. Those predefined types are the building blocks for user-defined types of tool data. After installation, the software engineering environment must be capable of providing services for user-level definition of tool data types. Additionally, the dynamic features of tool data must be considered.

According to the ANSI/IEEE Standard 729-1983, the difference between dynamic and static features lies in changes prior to or during execution time. Dynamic features of the tool data require that tool data types be defined during runtime of the software engineering environment system. If such facilities are not provided, the environment can only utilize a set of predefined data types such as in the operating system shell or in the language environment, where no dynamic type definition is supported. The criteria for differentiating the dynamic or static features are very much relative to the

underlying system. In the subject of tool data in a software engineering environment, the dynamic feature means that the types or values of the tool data can be defined and changed during execution. This issue is especially important to overcome the problems of a language-based environment; where all objects of the environment are subject to the type management mechanism of underlying language features.

True dynamic features can be achieved only when the software engineering environment supports type definition mechanisms and management tools for the object (tool data) in runtime. However, the claims of dynamic features in interpreter-based environments such as Common LISP or Smalltalk are not realistic when we consider the productivity of such system in the practice of software development [Narayanaswamy 1988]. The lack of a dynamic type definition capability in database-oriented environments or language-based environments makes it very difficult for such environments to achieve the goal of the software engineering environment until a schema or tool data type definition is dynamic. To achieve a productive, continuously evolving software development environment, the software engineering environment must support a sound mechanism for the environment user to define the new types of tool data and to utilize the environment system to manage the tool data in dynamic way.

3. OBJECT-ORIENTED DEVELOPMENT PARADIGM

The software engineering environment requires a well-defined mechanism to provide necessary data and procedural abstraction and modularity. The object-oriented design methodology supports those properties.

Object-oriented design is a methodology used to map the real-world problem domain to a representation of the solution domain in software. The object-oriented design approach is a unique method which provides all three properties necessary for the software design: data and procedural abstraction, information hiding, and modularity of software [Pressman 1987]. The object-oriented development paradigm is a good design technique that provides a systematic cohesion between data and operations.

The object management system (OMS) is a software system used to create objects and manipulate service calls from objects by sending messages so that both the structure and behavior of objects are managed in an uniform way. The object management system is different from an entity management system in its approach to building the structure and behavior of the system component.

3.1 Objects, Methods, and Messages

An object is an entity whose behavior and structure are described to represent a real-world component. The object is the basic unit of a system used to build another object in the object-oriented development. The object can be application software, documentation, file, display, string, or thing. The object consists of encapsulated data representing the private part of the data structure and operations that manipulate the encapsulated data. The object must have sufficient information to describe the nature of the object and how it can be manipulated. The private part of data in the object is the set of

internal variables manipulated only by operations of the object. The shared part of the object is viewed through the interfaces (specification of operations) that can be invoked when other objects send a certain message.

Booch describes the object as an entity that has the following characteristics [Booch 1986]:

1) an object has states that are persistent in time and space;
2) an object is characterized by the actions it permits and requires of other objects, namely, actor, agent, or server;
3) an object is an instance of some class;
4) an object is denoted by a name; and
5) an object is viewed by its specification.

Wegner describes the object as having a set of "operations" and a "state" that remembers the effect of operations [Wegner 1987b].

In the object-oriented paradigm, the objects are primitive elements that combine encapsulated data (private data structures), operations (methods), and properties (attributes) for data and procedures. The operations of the object are manipulated by means of messages.

The properties of the object consists of the information to describe the encapsulated data with persistent state and a set of operations to transform the state. The state of the data in the object is maintained after the operation has made some change. These characteristics are especially important when the object is an identity that will persist across several tools or applications.

The method is an operation that manipulates the private part of the data in an object. The method is typically a procedure or function that legitimately transforms the data structure. It is invoked by a message.

The message is a request to the object to perform one of its methods. The message contains the following information: the object to which the method is applied, the method requested, and the arguments to be passed to the method.

The details of the data structure and the implementation of methods are hidden and protected from the outside of the object. The modularity is naturally supported because the software system is grouped with component objects that have well-defined interfaces.

3.2 Classes, Instances, and Inheritance

Objects can be categorized by a general description. The description of a certain range of objects is a class, which serves as a template to be used to make new objects. In an object-oriented paradigm, every object is an instance of a class. Objects sharing the same class have the identical properties of an encapsulated data structure and a set of interfaces for operations. This organization by class enables type checking. For example, the object management system can provide strong typing to check the use and instantiation of every object in compile-time.

Inheritance is a mechanism by which an inheriting class is constructed by taking attributes, relations, and operations from the inherited class. Inheritable properties of a class are defined as a set of interfaces in the inherited class so that the visibility between classes is maintained. The inherited class is called the "superclass" and the inheriting class is the "subclass." The class hierarchy by inheritance groups objects by their properties.

The inheritance can be discussed in terms of specialization and generalization. Specialization is the union of properties from one or more

superclasses together with the ability to add properties. That is, inherited properties are shared by all subclasses. Generalization is the opposite of specialization; which takes the intersection of properties of type to build a superclass. In generalization, the inheritance will create a new class, which will be a superclass of already existing subclasses.

The most useful mechanism in software engineering is specialization, which can be used to define tool data types incrementally in lifecycle activities. Although generalization is useful for defining a frequently called type that is discovered after its subclasses are used, this kind of activity in a software engineering environment is not desirable because of potential inconsistency among objects.

3.3 Data Abstraction and Information Hiding

The term "data abstraction" refers to a technique that concentrates on essential characteristics of the data by hiding unnecessary information about the details of the data and providing associated functional characteristics. Data abstraction is accomplished when data typing is supported, in order to separate and hide the details of the data. Data abstraction allows the software designer to use the data in a correct and consistent way, and thereby focus on higher-level design activities.

The abstraction techniques have been developed since the first programming language was introduced. The procedural abstraction to abstract algorithmic information in design activities results in stepwise refinement and modularity. The abstraction techniques evolved with the software development practice of providing a suitable design methodology such as structured design or functional design methodology.

Abstracting the data object in software development became extremely important with more complex structures, and a set of dedicated procedures to manipulate them became necessary. To manipulate the complex data structure in a consistent manner, a set of associated procedures must be attached and related to the data to create a dedicated type. Data abstraction is a way to form such an abstract data type to couple the data object with associated procedures to hide the data structure and implementation detail.

The object-oriented design paradigm is well-suited for data abstraction because it encapsulates (hides information in) the data structure and implementation details of the operations (associated procedures), and provides the specification for the object in higher-level abstraction. The user of the object does not need to know how the operation takes place or how the data is transformed. With the aid of data abstraction and information hiding in the object-oriented design paradigm, it is possible to develop the software system in a systematic way by building component objects and integrating them.

3.4 Persistence

Persistence is the extent of the lifetime for data kept in the system [Wegner 1987a]. In the programming language environment, the lifetime of the data object usually does not exceed the lifetime of the program. Although data may be kept in the file after the execution of the program is completed, all other data used in the program are discarded. The parameters to the procedure or to a program has even shorter lifetime. Global data have longer persistence within the program's lifetimes. The database has a persistence that transcends the individual programs' lifetimes.

Persistence can be implemented in several ways: in the file, in the file with cache memory, and in the main memory. Conventional database systems implement the persistency by using a file system to store all changes of the data and to query the data with special operations. To improve the access efficiency, the caching mechanism has been introduced for persistent data objects in the file.

To provide the dynamically changing data in a multiply accessing environment such as a software engineering environment, persistence must be supported in the main memory level as well as in the file. The object of the object management system must be persistent in the main memory to provide an appropriate mechanism for tool data. The software engineering environment uses the persistent object as tool data interfacing among tools or with the user.

3.5 Language Supports and Strong Typing

There is a difference between abstract data in the programming language and the object in an object-oriented paradigm. The agent of the abstract data in programming language is the procedure call, which is handled by language facilities and implemented by binding the abstract data with appropriate parameters. The object is handled by the message that an agent sends to the object management system with object identity, method, and parameters for the method. Sending a message is not like a procedure call in the sense that the actual binding must be done at runtime, which creates more overhead than if it were done in the procedure call of the programming language. The identity of every object must be resolved at runtime and the type checked accordingly. Appropriate management of the objects requires a special environment, namely, object management system.

Object-oriented languages support the objects as language features, and the objects are instances of the class to provide the inheritance [Wegner 1987a]. The type-checking facilities for the object and message are provided by the language. Object-based languages such as Ada provide some degree of facility for object creation (package or task); however, objects are managed through the runtime support (Ada library). To fulfill the requirements of the object-oriented paradigm [Wegner 1987b], the programming language fails to provide all the facilities. For instance, to have an autonomous object that responds to the messages or operations and shares a state, the object must be a concurrent entity. However, the concurrency in a high-order language is generally not feasible. The class mechanism must be supported during runtime to provide dynamic organization of the objects while the language environment is not robust enough to support such type checking appropriately.

A strongly typed language provides a determination for type compatibility of all expressions for values at compile-time. Strongly typed language facilities support as many services as possible in a static way to help the programmer detect possible errors.

3.6 Object-Oriented Features in Software Engineering Tool Data

The object in software development can be identified in a data-flow diagram and modeled as abstract data [Booch 1986]. Modeling a real-world subsystem into an object is more natural than transferring it to a procedural and data-oriented entity because the object itself represents the real-world subsystem without structural transformation. The data-oriented development transforms the real-world system to a functionally decomposed entity.

However, it lacks the proper means of data abstraction and information hiding, concurrency, and persistent data.

To define an object that serves as abstract data among several tools in the software engineering environment, we need to focus on the design of the requirements of the tool data object in the environment with sufficient information to provide to the user. The object-oriented design technique is well-suited for such demands.

To provide the wide range of characteristics necessary for an ideal object manipulation, the object management system as a programming environment must support the most efficient method for object-oriented design of a software system. An object-oriented structure is suitable for the environment when problems of coordinating tools with the underlying host system exists, including hardware, languages, and operating systems [Cox 1986; Cox 1984]. This is because of its heavy dependence on extensive use of abstraction (either data abstraction such as the package or private types of Ada or program abstraction such as the generic package of Ada). The object as an instance of object type (class) is manipulated exclusively by operations (methods) that are encapsulated in their type. The implementation of these features varies. In a language, it can be done by means of symbol management in a static way. However, in a software engineering environment, it needs to be dynamic and managed by runtime facilities through some central repository (database or library).

4. EXPLICIT POLYMORPHIC TYPES FOR TOOL DATA

In a software engineering environment, the data manipulated by tools can represent many kinds of data objects such as code, documentation, graphic images, processes, and so on. These data objects are used by tools in the software development process. To support the smooth integration of tools, the software engineering environment needs well-defined mechanisms to represent the meaning of such data objects according to their usage and behavior. To categorize such objects and represent the meaning of those data, the type mechanism is introduced. The *environment types* used to categorize the tool data shared by tools can be of many forms including basic types such as integer, structured types such as record, array, or list, and process types such as the image of the program in execution.

This chapter first describes the required characteristics of an environment type in software engineering environments, and then discusses the *environment type processing language*. The environment type processing language includes two languages: *the environment type definition language* to define new environment types, and *the environment type manipulation language* to instantiate tool data from environment type and to operate on tool data by sending messages. A sample typed environment for software testing tools is described in a later section.

4.1 Explicit Polymorphic Types

In general, types can be described as sets of values with functions to obtain the values, from a universe of all possible computable values. If each value belongs to at most one type, the type system is monomorphic. If a value belongs to many types, the type system is said to be polymorphic. Operations on polymorphic types can be applicable to data of more than one type.

To support maximum reusability and productivity for the software development process, types should be defined as polymorphic and parametric, in which the parametric polymorphism allows the operations of types to work uniformly on a range of types so that the type construction process can be reusable. A type parameter used in the parametric polymorphism provides uniformity of type structure [Cardelli and Wegner 1985].

To provide good abstraction, information hiding, type inheritance, and parametric polymorphism for tool data types, the environment type definition language initially designed by Lindquist defines actual environment types for implementation and analysis [Lindquist et al. 1987; Levine 1988]. The environment type is defined *explicitly* by a definition code using an environment type definition language. Its type system supports the parametric polymorphism. We call such types explicit polymorphic types.

4.1.1 Environment Types in a Software Engineering Environment

In software engineering environments, environment types used for the tool data are abstract data types, which combine encapsulated data and operations to support an uniform behavior of types. This uniformity of behavior allows a consistent manipulation of type instances. Such types of objects are elementary software components to be reused in software development. Once the consistent manipulation of the tool data is supported by data abstractions, the use of the environment types supports reliable and modular software development as described in Chapter 3.

By using environment types for the software development, the new environment types of tool data are introduced for project specific purposes. To allow the evolution of software engineering data, types are not limited to a

finite number of initially predefined types. In addition to the predefined types, user definable types must be supported in the software engineering environment. The environment type definition must be dynamically supported to create the user definable environment types.

In the meantime, the predefined types can be used to construct a software engineering environment. This feature allows the software engineering environment system to be composed of the typed components. Examples of such types are structural node type, file node type, process node type, relation description type, and attribute description type of the CAIS environment.

4.1.2 Polymorphic Types for Software Engineering Tool Data

To provide reusability and modularity of environment type definition process, the environment type system must support polymorphic types. Polymorphic types for tool data allow the type system to maintain the same structure of types, which allows the reuse of operations for the range of types defined by type parameters. Reusability of environment data is most efficiently achieved by allowing the polymorphic types to define new types with an appropriate type parameter and by allowing inheritance of properties among types. The type system embedded in the software engineering environment must support this polymorphism with a type parameter.

4.1.3 Type Inheritance

The type inheritance is important for the new definition of environment types to provide for type evolution. A new type may inherit properties of super-types (parent types). Inherited properties are the content data structure, the attribute of the type, the relations of the type with other types, and the

operations. A new type can become a subtype of an existing type by inheriting every property of the supertype.

Multiple inheritance which allows the inheritance of the properties of more than one supertype provides an efficient way to create new types. The properties of one or more types are reused in the definition of a new type in such an inheritance mechanism.

4.1.4 Instances of Environment Type in the Software Engineering Environment

In a software engineering environment, the tool data are instances of the environment types. Some tools produce tool data and others consume them. In such an environment, the instances of environment types must be managed by the software engineering environment to provide appropriate services and maintain a consistent behavior for tools. Conversely, in a programming language, the instances of the type are owned and managed by a single program through a symbol table or run-time support.

To be shared among many tools, the persistency of tool data as well as their environment types in the software engineering environment is necessary. In such an environment, the tool data permit the tight integration of several related tools, such as in inter-application communication (IAC).

Each instance of a environment type in a software engineering environment is an autonomous entity capable of managing its encapsulated data over a period of time. That is, environment data are objects that are data abstractions with a set of operations and a hidden local state that persists.

4.2 Environment Type Constructor

The information shared among tools is described by an *environment type definition language*. Within this language are constructs for content,

attribute, relation, and operation. Name visibility for defining the environment type is accomplished by WITH and INCLUDES. Type inheritance is defined by SPECIALIZES. Type parameter is defined by the GENERIC construct. Type construction using APPLIES-TO provides parametric polymorphism. Types that are declared for the private purposes of a single environment type can be encapsulated by the NESTS construct. Figure 4.1 describes the syntax for the environment type definition language. The functions of each construct are described below.

generic <Generic Type Parameters>

The tool data are typed with parametric polymorphism through the GENERIC construct. The GENERIC clause allows the user of the environment type to supply actual type parameters when the environment type is instantiated. When the environment type is processed, the generic type parameters are bound to actual identifiers that are defined by **Generic Type Parameters.** The generic construct supports a parameterized definition of tool data with a great deal of reusability.

with <Ada Package Names>

The WITH construct is for the visibility of the Ada packages used for environment type definition.

includes <Environment Type Names>

The INCLUDES construct is for the visibility for the environment types used in the definition of the environment type.

nests <Nested Environment Type Definitions>

The NESTS construct encapsulates the definition of internally used environment types and makes them visible to the objective environment type definition. The NESTS construct allows the type designer to make a

```
e_type_definition ::=
        [generic (Generic_Name {, Generic_Name});]
        [with Ada_Package_Spec {, Ada_Package_Spec};]
        [includes Etype_Name {, Etype_Name};]
        [nests e_type_def_body {e_type_def_body} end nests;]
        e_type_def_body

e_type_def_body ::= application | e_type_body

e_type_body ::=
        environment type Etype_Name [specializes Etype_Name_List]
                [with    [contents_description]
                         [attributes_description]
                         [relations_description] ]
                [operation
                         {for Operation_Name use subprogram_spec }
                         {subprogram_spec} ]
        end Etype_Name;

application ::=
        environment type Etype_Name applies Etype_Name  to actual_parameters;

actual_parameters ::= parameter_association {, parameter_association}

parameter_association ::= [Generic_Name =>] Etype_Name

contents_description ::=
        contents
                Name : Etype_Name | Ada_type | Generic_Name {, Etype_Name | Ada_type |
Generic_Name }

attributes_description ::=
        attributes
                Name : value | type {, Name : value | type }

type ::= integer | float | boolean | string | date | list_type

value ::= value of one of the types of "type"

relations_description ::=
        relations
                Relation_Name to List_Of_Etypes [predefined]
                        [with attribute attributes_description ]
                                [cardinality = range] [primary]
                { Relation_Name to List_Of_Etypes [predefined]
                        [with attribute attributes_description ]
                                [cardinality = range] [primary] }

subprogram_spec ::=
        procedure Name [formal_part]; | function Name [formal_part] return type_mark;

formal_part ::= (parameter_spec {; parameter_spec} )

parameter_spec ::= identifier_list : mode type_mark

mode ::= [in] | inout | out

type_mark ::= Etype_Name | Ada_type | Generic Name
```

Figure 4.1. Syntax of Environment Type Definition Language

modular definition for a substructure of the environment type while the details of the definition are hidden from the user of the type. The properties of the environment type in the NESTS construct are inherited by the objective environment type. However, this mechanism is not for the type inheritance since the NESTed types are not part of the environment types sharable in the software engineering environment. The NESTS construct is not transitive, so that only one level of nesting is allowed.

specializes <Environment Type Names>

Type inheritance is supported by the SPECIALIZES construct to allow inheritance of all properties of content data structures, attributes, relations, and operations. This inheritance does not permit visibility of definitions. SPECIALIZES allows multiple inheritance.

applies <Environment Type Name> to <Actual Type Parameters>

The APPLIES-TO construct allows the generic instantiation of the environment type using type parameters. The effect of this construct is to create new types of the actual type parameter set and bind them to the body of the environment type definition or to the nested type definition. With the GENERIC construct, APPLIES-TO completes the parametric polymorphic type definition mechanism in the software engineering environment.

contents

The CONTENTS construct defines the encapsulated data in the environment type that carries information among tools. The data structure that the environment type object carries is stored in the content and used by the tools through operations applied on them. The content can be described by the environment type, Ada type, or generic type name. Multiple content fields can be defined to construct a record.

attributes

ATTRIBUTES define the association of information about content and relations of the environment type. For example, an attribute of the environment type can be the range of the value for the content or creation information of the environment type. ATTRIBUTES describe the information about the type or object but are not used to carry data for the environment type.

relations

RELATIONS is a mapping among instances of environment types. The relation of environment types associates two different entities in the entity-relationship model. The relation is a channel for the tool to navigate other instances of environment type.

A relation is PREDEFINED if the environment type forces the tool to access the relation only by defined operations. The relation can have a CARDINALITY that limits the number of relationships to other instances. Cardinality defines the number of relationships emanating from the instances of the environment type. Each relation is assumed to be secondary unless it is explicitly defined as PRIMARY. A relation can have attributes that attach informations to the relation; these attributes are in the same format as the ATTRIBUTES construct.

operation

The OPERATION construct defines the methods used to manipulate the information declared in the environment type. The operations are the only means to manipulate any of the encapsulated data structures in the type instances. The operation in this environment type has the Ada subprogram specification, that is, procedures and functions. A FOR-USE construct allows

the environment type to override any inherited operation and to redefine new specifications with the same operation name.

4.3 Environment Type Instances

The environment type and the tool data must be consistently managed by the software engineering environment. These environment types and tool data belong to not one but many tools in the software engineering environment.

There are two logical worlds in environment type management: the definition world and the instance world. The tool data are realized by instantiation from the environment types. The environment type definition world is a conceptual area where the environment types are organized in a library. The environment type instance world is a conceptual area where the tool data are stored. The tool data are created by a tool and used by other tools to store, query, retrieve, and carry information among tools. These tool data are shared objects that are persistent in a software engineering environment. Figure 4.2 shows the inter-relationships among the environment types and tool data as instances of the environment types among tools.

4.3.1 Environment Type Instantiation

An environment type is instantiated to create a new object of tool data. Once instantiated, the tool data from one environment type share the same properties but carry their own data values. Environment type instantiation must support the construction of a data structure along with appropriate attributes and relationships for the object to carry the information.

The *environment type manipulation language* is used to provide mechanisms to create the environment type instance (tool data). The environment type is instantiated by the DEFINE construct. DEFINE creates the instance of the named environment type in a given pathname with a

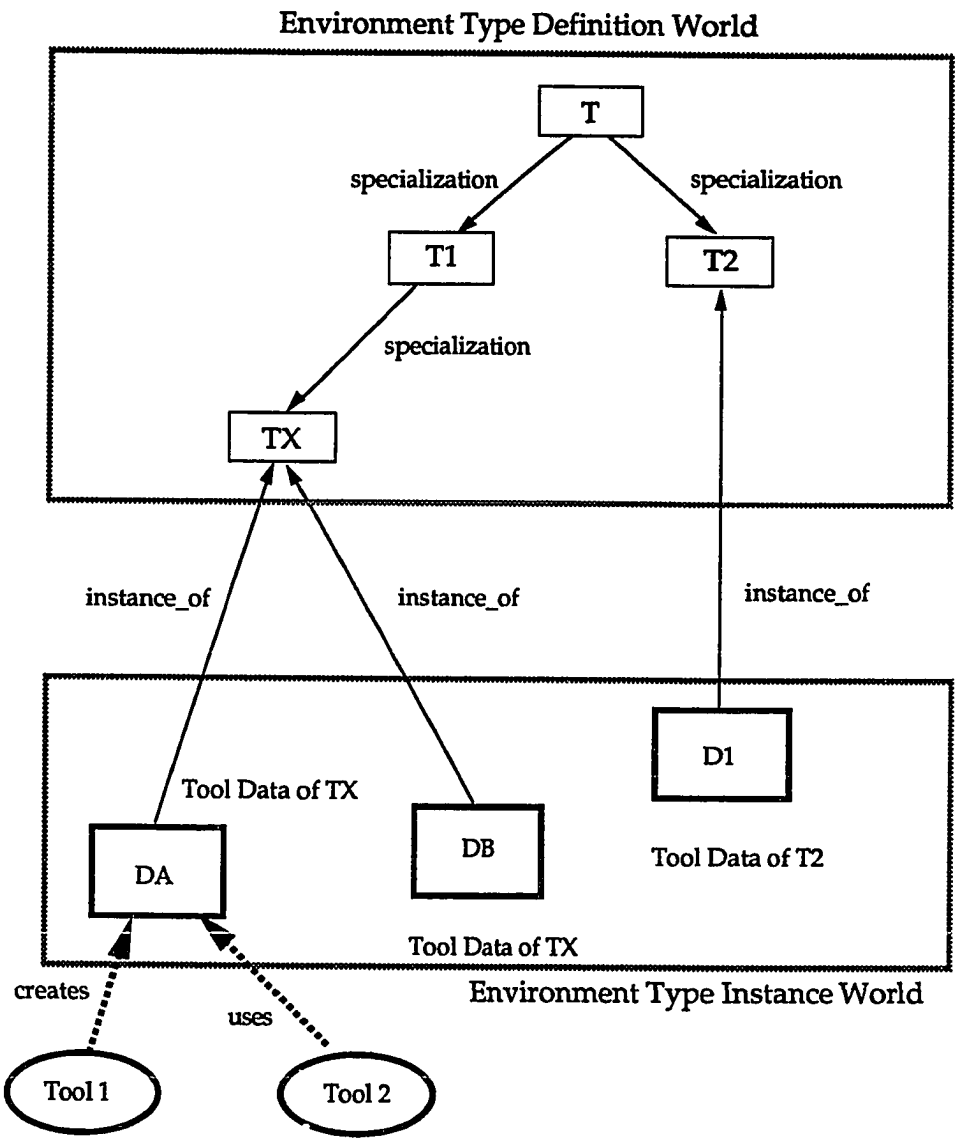


Figure 4.2. Environment Types and Their Instances among Tools

specified intent. If there is an instance for the environment type with the specified pathname, the instance is opened for the given intent.

After DEFINE, a tool uses such an instance by sending messages. The SEND construct sends a message to the specified instance, which becomes tool data when it is used by the tools. The CLOSE construct closes the instance. After CLOSE, the tool cannot access the tool data.

The syntax of the environment type manipulation language for instantiation and use is described in Figure 4.3. The DEFINE, SEND, and CLOSE statements are used in the tool code written in the host language.

The *node handle* for the instance is the identifier used to access the tool data in the software engineering environment. The node handle defined in the tool code for a specific tool data can differ from one tool code to another. The actual reference to the tool data is resolved by pathname not by the name of the node handle in tool code.

The DEFINE statement provides the intent of use of the instance. Among tools, the intent information is vital so that mutual exclusion and locking mechanisms can operate safely.

The INCLUDE construct provides visibility of the environment type used in the tool code.

4.3.2 Persistence of Environment Type Instances

The tool data must be a persistent object that keeps the state of the value. The persistency of the tool data must be supported by both the file system and the main memory because the tool data can carry an actual process that can be shared by several tools. The CAIS node model provides a proper mechanism to support persistent tool data. The operations for tool data that are defined

```
include Etype_Name {, Etype_Name} ;

define Node_Handle : Etype_Name ( Pathname , Intent ) ;

send Node_Handle . Operation_Name [ actual_parameter ] ;

close Node_Handle ;

Pathname ::= CAIS pathname

Intent ::= CAIS INTENT description

actual_parameter ::= Ada Actual Parameter Description
```

Figure 4.3. Syntax of Environment Type Manipulation Language

in the environment types must be persistent and active during the lifetime of the tool data in the software engineering environment.

4.4 CAIS: A Building Block for an Environment Type System

The CAIS is used in the research as a basic software engineering environment to build the type system for tool data. It provides a set of interfaces among APSE (Ada Programming Support Environment) tools and the virtual operating system. It provides reasonable source-level portability to other environments. The structures and functionalities of the CAIS related to this research are considered below.

4.4.1 Architecture of APSE

The architecture of the APSE can be described as a set of layers from the host machine to the user. The core of the APSE is the KAPSE (Kernel APSE) that provides a set of transparent interfaces to host machines and operating systems with a common set of capabilities. The next layer, the MAPSE (Minimal APSE), includes the set of software tools used to support a minimal set of functions for software development. MAPSE tools include compilers, editors, and linkers. MAPSE tools interface through KAPSE. The highest layer of an APSE provides project-specific tools and services such as design, specification, testing, and documentation. Figure 4.4 shows the APSE architecture [Kramer et al. 85; Buxton 1980].

4.4.2 CAIS

The CAIS is designed to support full functionality for the APSE. The CAIS was proposed to maximize the transportability of the APSE tools and environment objects as well as the other environment tools among different host machines and operating systems [Kramer et al. 1985]. The CAIS defines a set of interfaces that are universally useful so that the user of the APSE uses

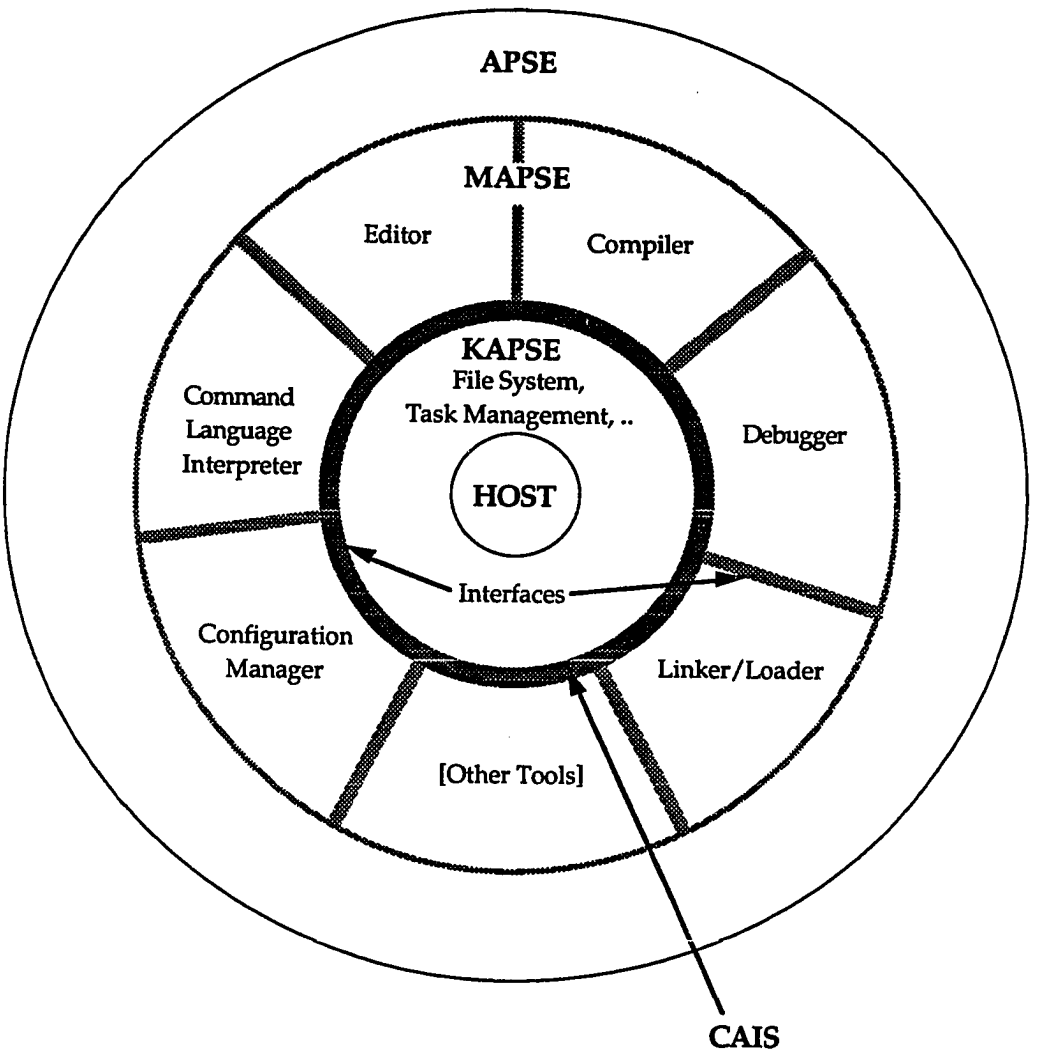


Figure 4.4. APSE Architecture [Source : STONEMAN]

the CAIS interfaces to achieve portable and methodology-independent development. The implementation details of the interfaces to host systems are well encapsulated and abstracted. The CAIS is an important building block for portability and enhanceability in evolving Ada development environments.

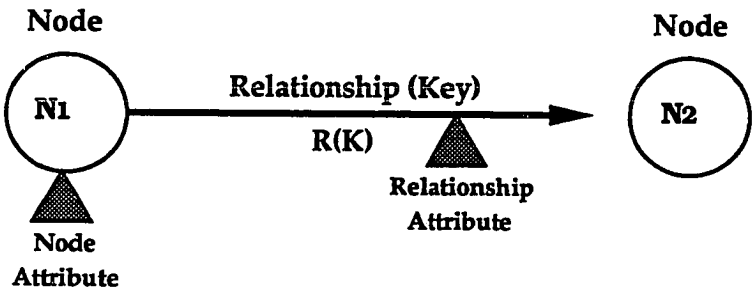
The CAIS is based on the entity-relation-attribute model (ERA model). This model supports the modeling of environment objects by entity and mapping among the objects by relations. The properties of the entity and the relations are described by attributes attached to them.

The CAIS consists of three major interfaces: node model, process management, and CAIS input and output management. We shall review these subsystems and the revision of CAIS with respect to the typing facilities as they relate to the research concerns.

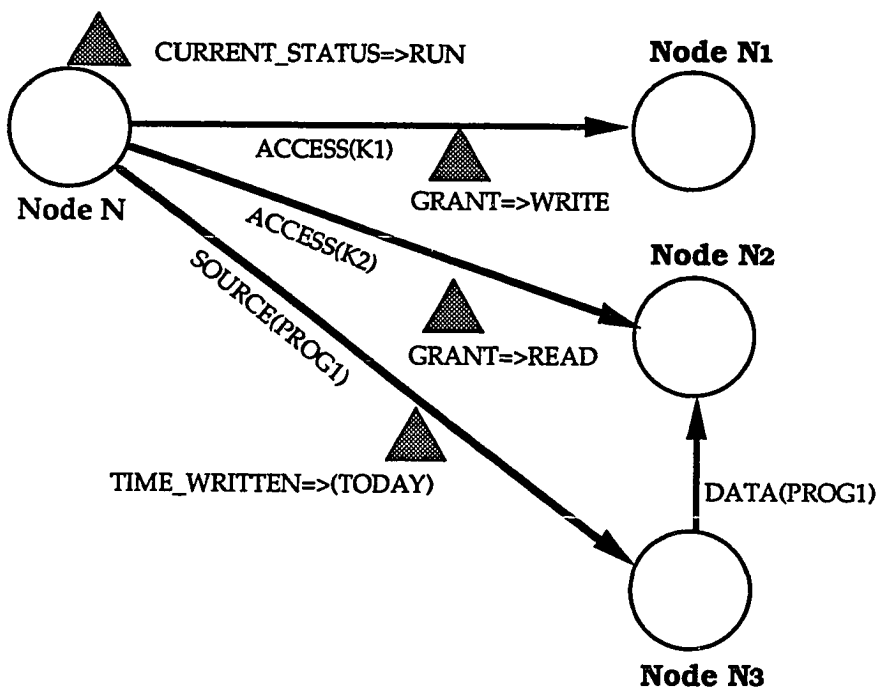
4.4.3 Node Model

The CAIS is structured by an ERA model called the node model, which structures entities that are objects of the environment, with associations among entities and their attributes. The relations connect the entities related to each other. The attributes carry the information about the entities and relations.

Each entity in the node model is represented as a node. A *node* is a data representation of the information about the entity. The basic schema of the node model is shown in Figure 4.5 (a). A node can be represented as an object if the modeling provides an object-oriented model. In the ERA model, the focus of the structure is on the identity of the entity and the relationships among entities.



(a) Schema for Node Model



(b) Example of Node Model

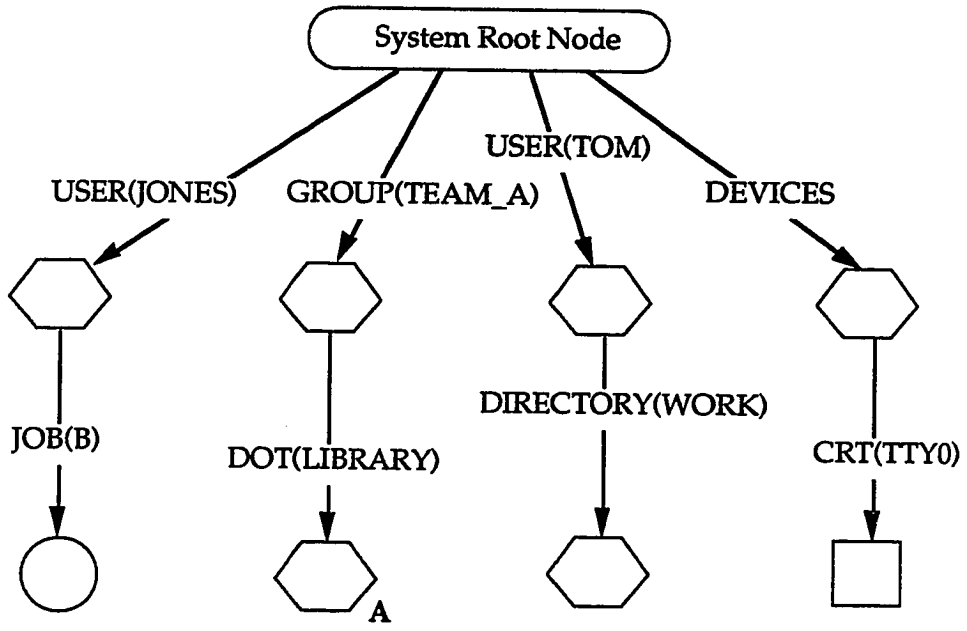
Figure 4.5. Node Model

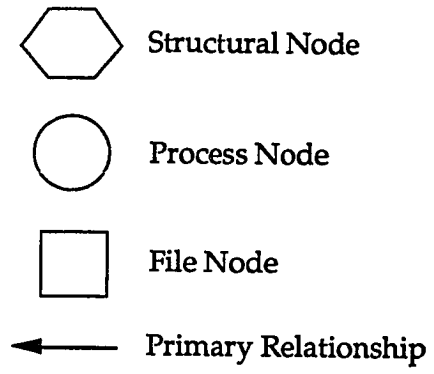
A node is categorized as a structural node, process node, or file node. A *structural node* represents a user, group of users, directory, and roles. It groups the other nodes and represents meaningful organization for node structure. A *process node* represents a process in the environment and provides information about the process such as the status of the process. The actual code image of the running process represented by the process node is stored in a separate file represented by the corresponding file node. A *file node* represents an external Ada file and device.

Interconnections among nodes are represented by *relationships* that are the specific relations with relationship keys. These keys distinguish the relation by a unique named value. The relationship allows the node to access other nodes by navigating relationships emanating from the node. An example of a simple node model is shown in Figure 4.5 (b). With both primary and secondary relations, the node model furnishes a hierarchical model with the capability of network structure. If the relation is primary, the node model is constructed as a tree structure (node tree). Among the nodes in the node tree, the secondary relations map the node model as a network structure. The concatenated association by relation name and relationship key provides at least one unique *pathname* to reach a specific node from either the system root (full pathname) or the current node. Figure 4.6 shows an example of hierarchical node structure and pathname.

The properties for the nodes and relations are represented by *attributes* attached to them. These attributes are used to store the data that describe the information about the nodes or relationships.

The relation name and attribute name are provided either as predefined or user-defined. With the set of interfaces to handle the node





Node A can be referenced by pathname 'GROUP(TEAM_A)'LIBRARY

Figure 4.6. Example of Hierarchical Node Structure and Pathname

model, the node model can be organized to create a project-oriented framework that may involve people, tools, activities, and data in a natural way [CAIS 1986; Kramer et al. 1985; Lindquist 1988].

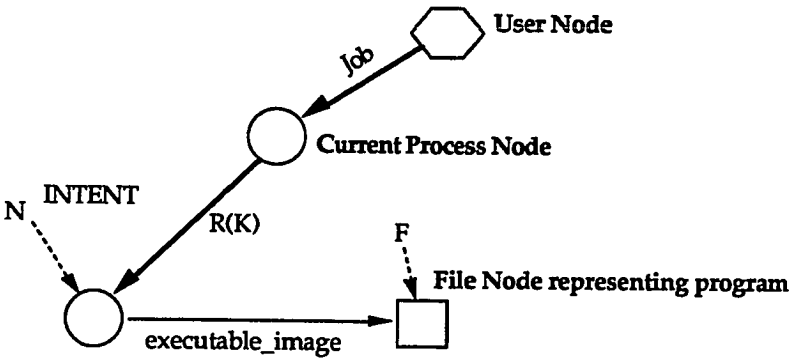
4.4.4 Process Management

The process is represented as a CAIS process node and provides information about the running process in the node model. The CAIS interfaces support the management of processes by spawning processes, invoking processes, and creating jobs. Other services to suspend, resume, await, and abort are supported. The necessary relationships and attributes are set while the service routines are performed. The Figure 4.7(a) shows a typical node structure after process creation.

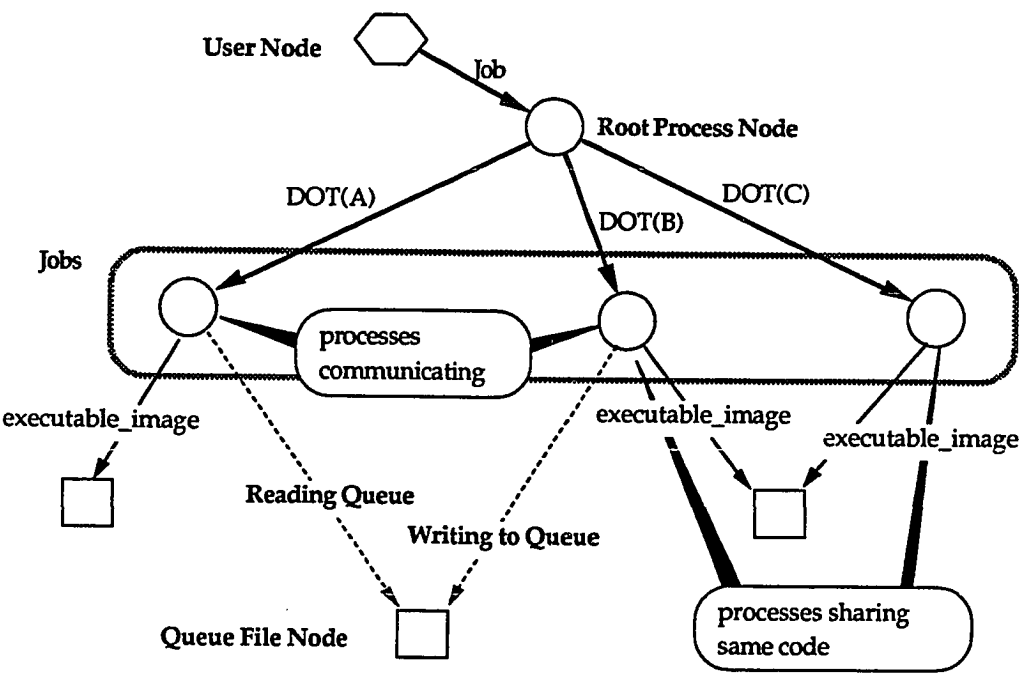
The interprocess communication is supported by a piping mechanism that employs queue I/O. Process synchronization is provided by await. The solo queue is a file node whose content holds the data written by writer processes and read by reader processes. The appropriate order of write and read operations is managed by the CAIS. In this research, the solo queue model is used to provide interprocess communication. Figure 4.7(b) shows such mechanisms. Details of the other queue models such as copy queue and mimic queue can be found in [CAIS 1986; Kramer et al. 1985].

4.4.5 Typings of CAIS Node Model of CAIS 1838-A Revision

The node model in CAIS was revised in CAIS DOD-STD-1838A incorporating the concept of typing. The nodes, relationships, and attributes are conformed by type definitions. All type definitions are modeled with a node structure. The required relationships and attribute details are forced to be created when a certain type of node is created by the typing. The objects of the type in CAIS are persistent. The CAIS revision-A allows mechanisms for type change after



(a) After SPAWN_PROCESS (N, F, INTENT, K, R) is performed



(b) Process Nodes, Jobs, and Interprocess Communication

Figure 4.7. Process Node Structure

the instances of the types are created. Possible inconsistency caused by the type change may occur among objects of the type and the operations on them. Specialization of the type is suggested to remedy this problem.

General requirements for the management of the types and objects are not sufficiently specified in the CAIS revision-A, in which it is suggested that the node model is to be typed and managed. However, the management of the objects is not type-based, so it is the user's responsibility to correctly create and manipulate the objects. The user-definable type is not supported, and only the predefined types for the nodes, relationships, and attributes are available. Type evolution by specialization is not flexible or rich enough to provide typings for the various kinds of tool data in the software engineering environment.

The CAIS provides a well-structured interface for building a type system. The node model and process management support versatile tools for a concurrent, persistent, and modularized environment for the type system. We built the type system on top of the CAIS in this research.

4.5 Example: Software Testing Environment

This section demonstrates the capability of environment type definition for software testing during the software development cycle. Software testing is a complex process that involves many tools and data. The tools include a processor to formulate a software specification, a test generator to produce test programs and corresponding test data such as IOGen (which produces a token list from a syntactically correct source code), and a test oracle to execute the test program with the test data [Lindquist and Jenkins 1988].

The tool data include the specification, the test objectives, the test program, and the test data. These tool data are shared by the testing tools such that one tool generates the data while the others use the data.

Tools for the Testing Environment

Our example consists of two tools: the Test Generator and the Test Oracle, as shown by the schematic diagram in Figure 4.8. The human tester runs the software and monitors the activity interactively. The tester enters the specification of the modules to be tested, selects the desired actions of the test generator via an interactive input device, and records the results.

Test Generator

The Test Generator has three inputs:

1) the Source Program in the form of a syntax tree and a symbol table generated by tools such as IOGen's scanner/parser (or in a form determined by a compilation system),
2) the Specification in a structured form to contain the functional requirements of the target software such as input requirements, exception handling, and output requirements, and
3) the human interaction from the tester, who interactively selects the desired action that is converted into an appropriate predicate form by the tester.

The action part in predicate form is attached to the test objectives of the given target software to generate the Test Program and the Test Data. The Test Generator analyzes the target software, generating input and output assertions in the form of a predicate tuple list. The next step selects tests from the list and combines the desired output from the specification to form a test case. The Test Objectives are a list of tuples of the following three elements:

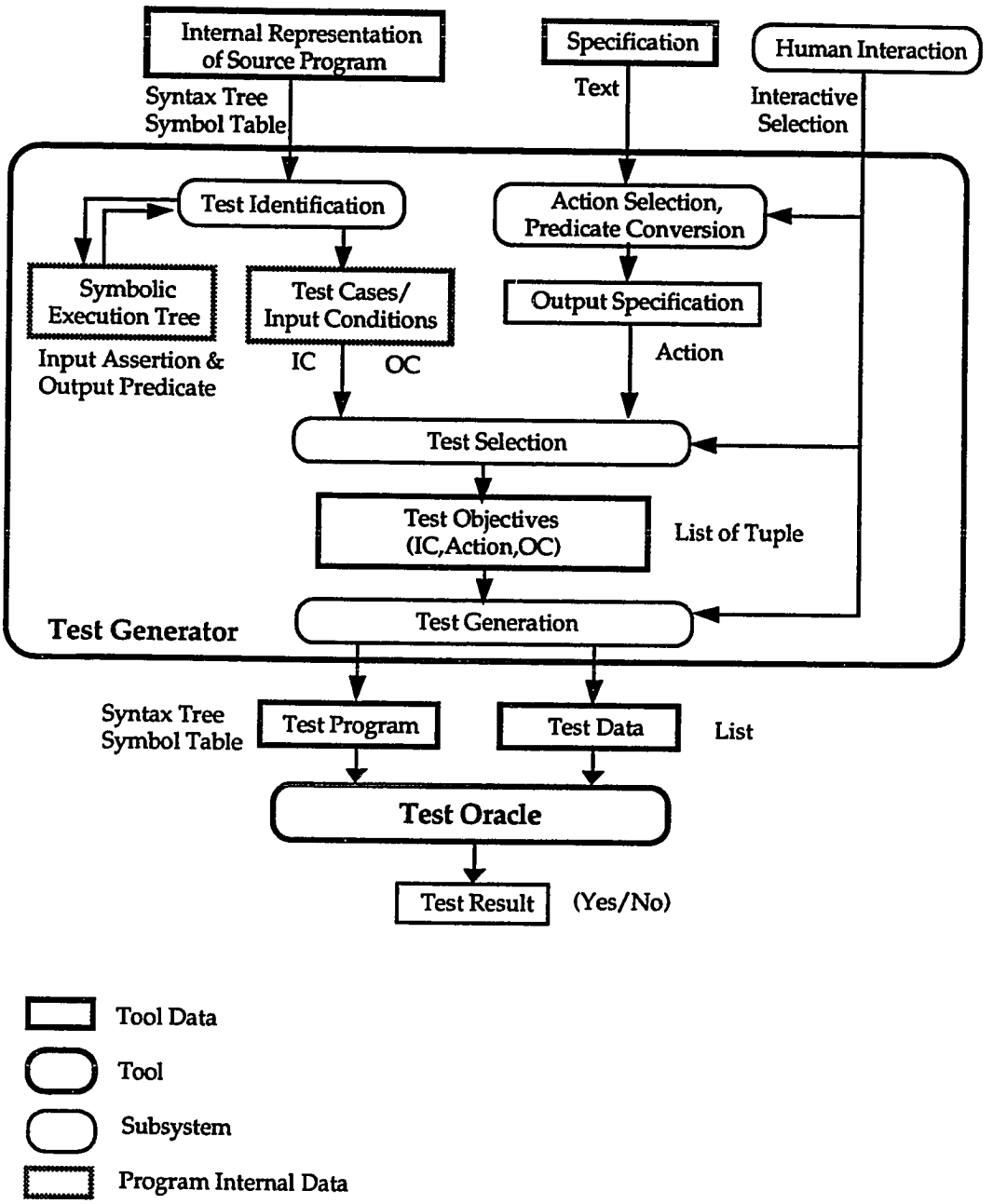


Figure 4.8. Testing Environment Schematic Diagram

Input Condition (IC), action (ACTION), and Output Condition (OC). IC is the input assertion for the module ACTION, and OC is an assertion describing the expected output. The Test Generator generates the Test Program for the selected action of the target software using IC and ACTION and produces the Test Data for the test program according to the specified input requirements with human interaction.

Test Oracle

The Test Oracle is an automated tool that decides the correctness of the target software by executing the test program. The Test Oracle executes the test program with the test data and decides the correctness for the given test case. It takes two inputs: Test Program and Test Data. It will execute the internal form of the test program using the test data.

Tool Data for Testing Environment

This section turns from the testing process to a description of environment types for the data shared by these tools. Figure 4.9 shows the entity-relation diagram for these types.

The tool data Specification, Source Program, Test Objectives, Test Program, and Test Data correspond to the entities SPECIFICATION, SOURCE_ENV, TEST_OBJECTIVE, TEST_PROGRAM, and TEST_DATA_ ENV, respectively. The contents of each type are included within a rounded rectangle. The circled numbers describe the typical sequence of the creation of an instance for the environment type. The relations among the types are described as directed arcs. The relations shown here are a minimal set to relate the relevant tool types. Appendix A shows these type definitions in detail using the environment type definition language described in the previous section.

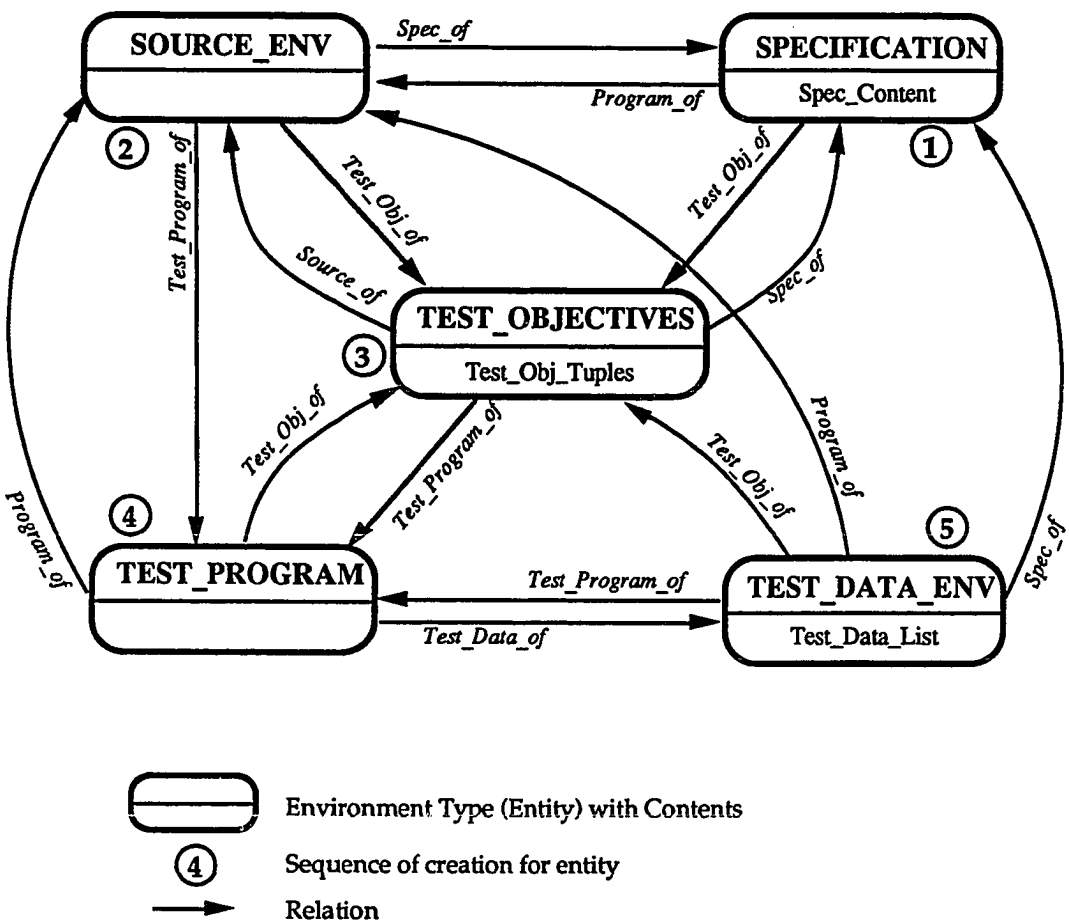


Figure 4.9. Testing Environment Entity-Relation Diagram

Specification

The SPECIFICATION contains text that describes the requirements of each action for a given module of the source program. Attributes such as the number of items, the resource tool used to generate the specification, creation date, update date, or size define the necessary information about Specification as tool data. Relations link Specification components such as SOURCE_ENV or TEST_OBJECTIVE. Since a specification can have multiple source programs or test objectives, the cardinality of these relations ranges from 1 to many. The operation part of SPECIFICATION supplies all necessary actions to link relations to other tool data explicitly and to query the information encapsulated within the type.

Source Program

SOURCE_ENV inherits the contents, attributes, relations, and operations from the generalized PROGRAM_ENV. It shares the contents and attributes with supertype PROGRAM_ENV. However, it adds other special relations emanating to SPECIFICATION, TEST_OBJECTIVE, or TEST_PROGRAM.

SOURCE_ENV shares most of the properties with TEST_PROGRAM. The supertype PROGRAM_ENV defines the common properties of contents of entities such as SYMBOL_TAB, which is another environment type SYMBOL_TABLE. It also defines SYNTAX_TREE, which is defined inside of PROGRAM_ENV type definition by the NESTS construct. When PROGRAM_ENV imports the operations from the environment type SYMBOL_TABLE, it redefines the operation of ADD_SYMBOL to INSERT_TOKEN with a new parameter list. This change affects only this type and its subclasses.

Test Objectives

TEST_OBJECTIVE contains the list of tuples of input condition, action, and output condition (IC, ACTION, OC). This list is a generalized list type using CAIS_LIST_TYPE, which is defined in the CAIS. The TEST_OBJECTIVE is used to store all possible test cases of the given target software. It stores the input condition and output assertion for each test case produced by a symbolic execution tree [Hantler and King 1976]. It is a database to be referenced when the test program and test data are generated by an interactive command from the human tester.

Test Program

TEST_PROGRAM is a specialized type of PROGRAM_ENV that shares its contents, attributes, and relations. The test program for a given ACTION is generated from a human command based on TEST_OBJECTIVE with the desired output specification supplied from SPECIFICATION. The reference to SPECIFICATION can be navigated through TEST_OBJECTIVE or SOURCE_ENV indirectly since both types have unique SPECIFICATIONs.

Test Data

TEST_DATA_ENV contains the list of test data corresponding to the parameter list for the given test program. It can have multiple sets of test data for a wider test range. The TEST_PROGRAM allows multiple test data sets by using more than one relation pointing to TEST_DATA_ENV.

Summary

By defining types for tool data among various tools and methods of using them, it is possible to manipulate sharable resources with greater reliability in a software engineering environment. The incremental definition of tool data can be achieved by this modularity and by greater effort in the design phase.

Both the specialization mechanism and dynamic visibility control provide greater flexibility in tool design. The entity relation model with the object-oriented paradigm in environment typing mechanisms greatly enhances the reusability and modularity of software engineering environment data.

5. DESIGN OF ENVIRONMENT TYPE MANAGEMENT SYSTEM FOR SOFTWARE ENGINEERING ENVIRONMENT

This chapter describes the details of the design process of the environment type system. Our design is based on the CAIS environment although an environment type system can be designed and implemented in various environments. The CAIS was chosen as an underlying system because it supports entity management capability, persistency of objects, and concurrency, portability, and most of all because it is operational. Under contract with U.S. Department of Defense, the operational definition of CAIS is now running in the Computer Science Department of Arizona State University. Ada was selected as a host implementation language because it has powerful supports for modular development, tasking, and packaging, and for its good PDL (Program Design Language) functions.

The type system performs various functions in the management of environment types, tool data, tools, and library. The management of environment types and tool data is performed by the **environment type management system**. This chapter describe the architecture and data flow of the system. The focus of this chapter is to identify subsystem components and their functions in order to build the environment type system in a dynamically evolving environment.

5.1 System Architecture

The environment type management system is built on top of the CAIS as illustrated Figure 5.1. The environment type management system manages environment types and tool data. The type builder writes the environment type definition code to create new environment types in a software engineering environment. The tool builder writes the tool code in the host

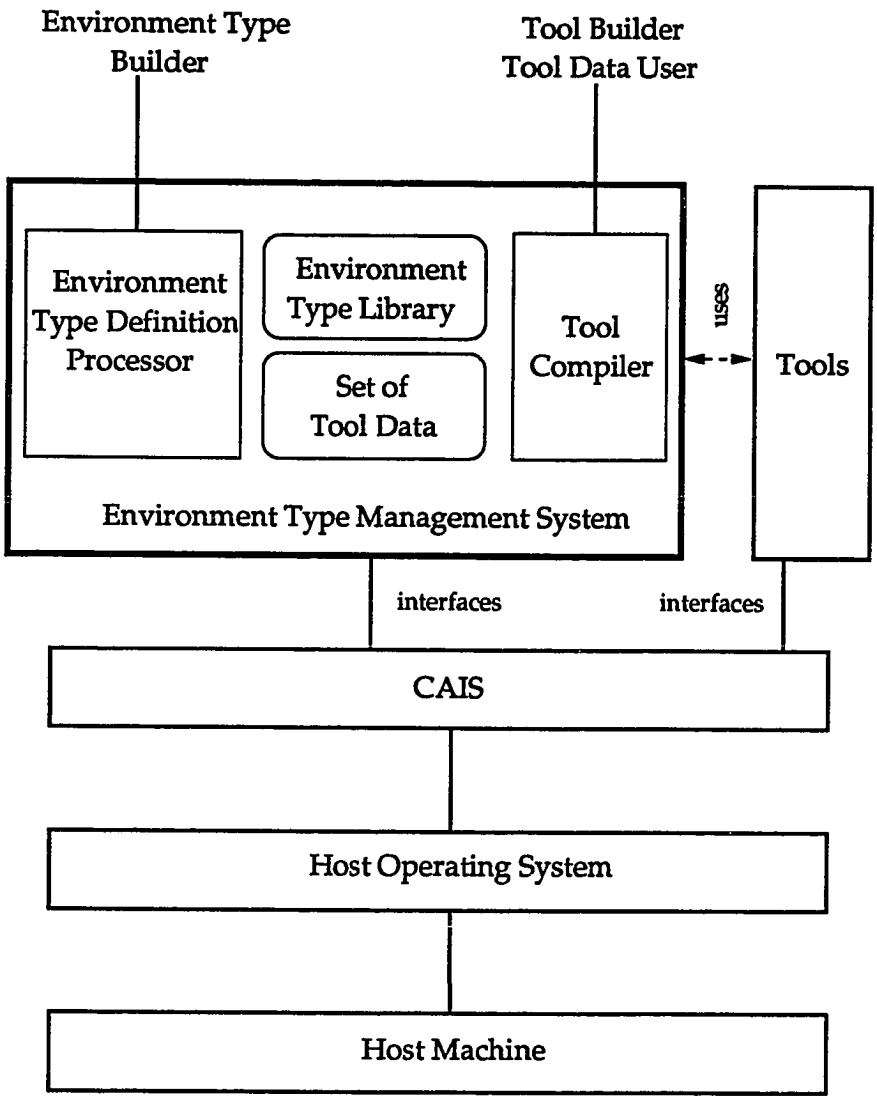


Figure 5.1. Architecture of the Environment Type System

language. The environment type builder uses the environment type definition processor (ETDP) to process the environment type definition code. The tool writer uses the tool compiler to process the tool data management commands in tool code. The environment type definition processor and tool compiler are the basic tools used to process the environment types. The environment type library and the set of tool data are managed by the environment type management system.

5.1.1 Interrelationships of Tools and Tool Data

The environment type management system is an integrated system that provides capabilities to define and catalogue new environment types, instantiate the tool data from environment type, and manipulate the tool data from the tool code in a consistent way. The environment type management system allows both object management and entity management. Figure 5.2 shows the interrelationship among various entities of the environment type management system.

ETDP (Environment Type Definition Processor) catalogues new environment types into a dynamic environment type library written in the environment type definition codes.

The tool compiler is a preprocessor that checks the validity of the use of tool data according to their environment types as used in the tool code. The tool compiler, then, generates code for proper tool data manipulation. The ETLM (Environment Type Library Manager) manages the environment type library to provide the necessary services for the environment types. The conceptual domain of the environment type definition structures is called an *E_Type Definition World*. The tool data instantiated from the environment types are stored in the node model of CAIS. The TDM (Tool Data Manager)

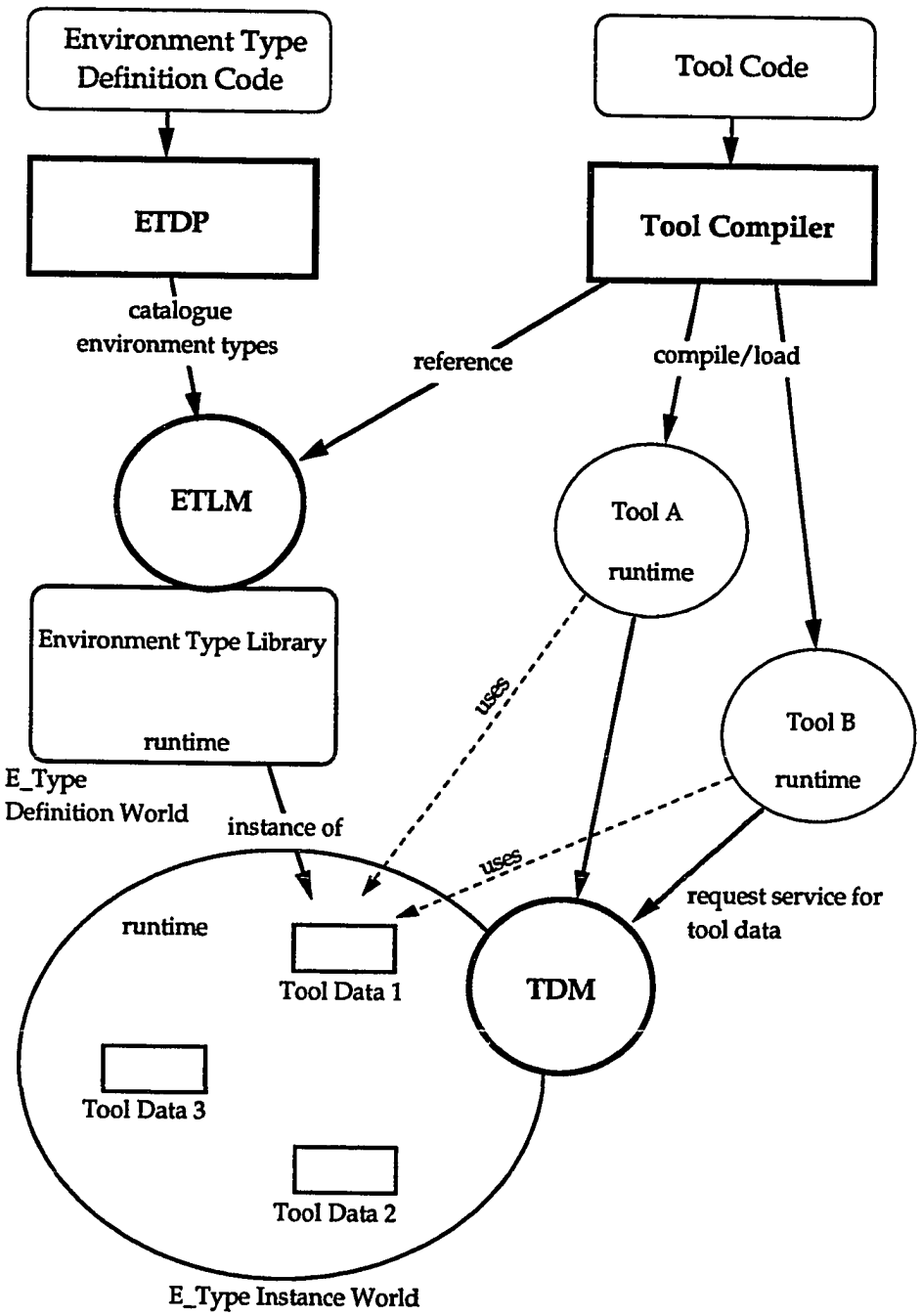


Figure 5.2. Interrelationship Among Entities of the Environment Type Management System

provides necessary services for the tool data. The conceptual domain of tool data structures is called an *E_Type Instance World*.

5.1.2 Dynamic Environment for Tool Data

During the execution of the environment type management system, the user defines the environment types and instantiates the tool data from existing environment types. The software engineering environment must be able to support the environment type management system as its subsystem and provide runtime capability to manage the environment types and tool data.

The environment types and tool data are objects that must be available to many tools (sharable) and are still available after execution of the tools (persistent). The ETDP and tool compiler are dynamic tools that run in the software engineering environment. The environment type library providing necessary type informations must be managed in a dynamic way to support both environment type definition and tool data instantiation.

Once the environment type definitions are processed and catalogued in the environment type library, another environment type can be defined by inheriting properties of the environment types. Tool data are instantiated from the catalogued environment types. Every activity, from environment type definition to tool data use, must be done in a dynamic way. This requirement is necessary to provide true reusability in software development where users are free to define new environment types and use them to create tool data and share them among various tools.

All objects, including tool data and environment types, must be sharable and persistent among tools. To support efficient use of computing resources and productive development of the software, it is best that tools be the processes that are subject to the scheduling mechanism of the host

operating system. Yet the tools as processes must be able to share tool data while they are executing.

5.1.3 Object Management System and Entity Management System

The environment type management system is an object management system (OMS) that treats every entity as an object. It is capable of object creation, class hierarchy, class inheritance, persistence, message management, and method application.

The environment type management system is also an entity management system (EMS) that manages all entities of the software engineering environment in an entity-relation model. In this capacity, it is capable of entity identification, relation definition among entities, and attribute management.

The object of the software engineering environment can be viewed as an entity for EMS while the same object can be defined as an object for OMS. The environment type management system provides characteristics of both object management system and entity management system.

5.2 Design of Environment Type Management System

The design process attempts to reveal desirable characteristics of the environment type management system. The graphical notations used in this chapter are based on the pictorial system description techniques of Buhr [Buhr 1984].

5.2.1 Requirements

Software engineering environment users generally fall into two categories: environment type builders (environment type suppliers) and tool builders (environment type users). The two distinct roles result in different activities of environment type creation and environment type use. Accordingly, there

are two distinct input sources: environment type definition code and tool code.

An environment type definition code has a specification part and an operational definition part (body) using the environment type definition language (described in Chapter 4). Tool code includes tool data manipulation commands that use environment type management services. The tool data are the instance objects of the environment types.

Environment type definition code and tool code are processed in the environment type management system. The environment type definition library stores the environment types and is managed by the environment type management system. The tool data are created and used in tool code; they are dynamic and persistent objects managed by the environment type management system.

The actual operations for manipulating tool data take place when the tool codes are run. Any request for tool data operation is through a message from the tool to the environment type management system, which handles the message to initiate appropriate operations to the tool data. The set of operations applied on the tool data is shared among all tool data of the same environment type. A proper mutual exclusion among various calls for operations from many tool data must be provided.

The environment type definition code and tool code are processed during runtime of the environment type management system. The generated tool must be able to access tool data during runtime of the tool.

The environment type library must be accessed only through the environment type management system. The environment type library is structured as a node model of CAIS. Although the behavior and structure of

the environment type library can be implementation specific, all necessary protection and query mechanisms must be provided. The tool data must be autonomous, sharable, and persistent.

The run-time management of tool data is supported by object management facilities of the environment type management system. The type checking mechanism of the tool compiler references the environment type library that is available during runtime of the CAIS.

5.2.2 Data Flow Diagram

The environment type management plays a central role in managing the services and activities for tools using environment types and tool data. The data flow diagram shown in Figure 5.3 describes high-level data flow among several entities of the typed software engineering environment. It shows only related entities such as the environment type supplier (environment type building activity), the environment type user (tool programming), the environment type management system, host language tools (Ada compilers, linker, load and dispatch facility), and tools generated from the tool program that include use of environment types by tool data. The environment type library serves as a central repository for reusable environment types. The tool data are conceptually grouped and managed by the environment type management system.

Using the environment type definition language, an editor creates the environment type definition code, which is processed by the environment type management system to generate type structure and then catalogued into the environment type library. The environment type management system also generates environment type code that is host language (Ada) compatible

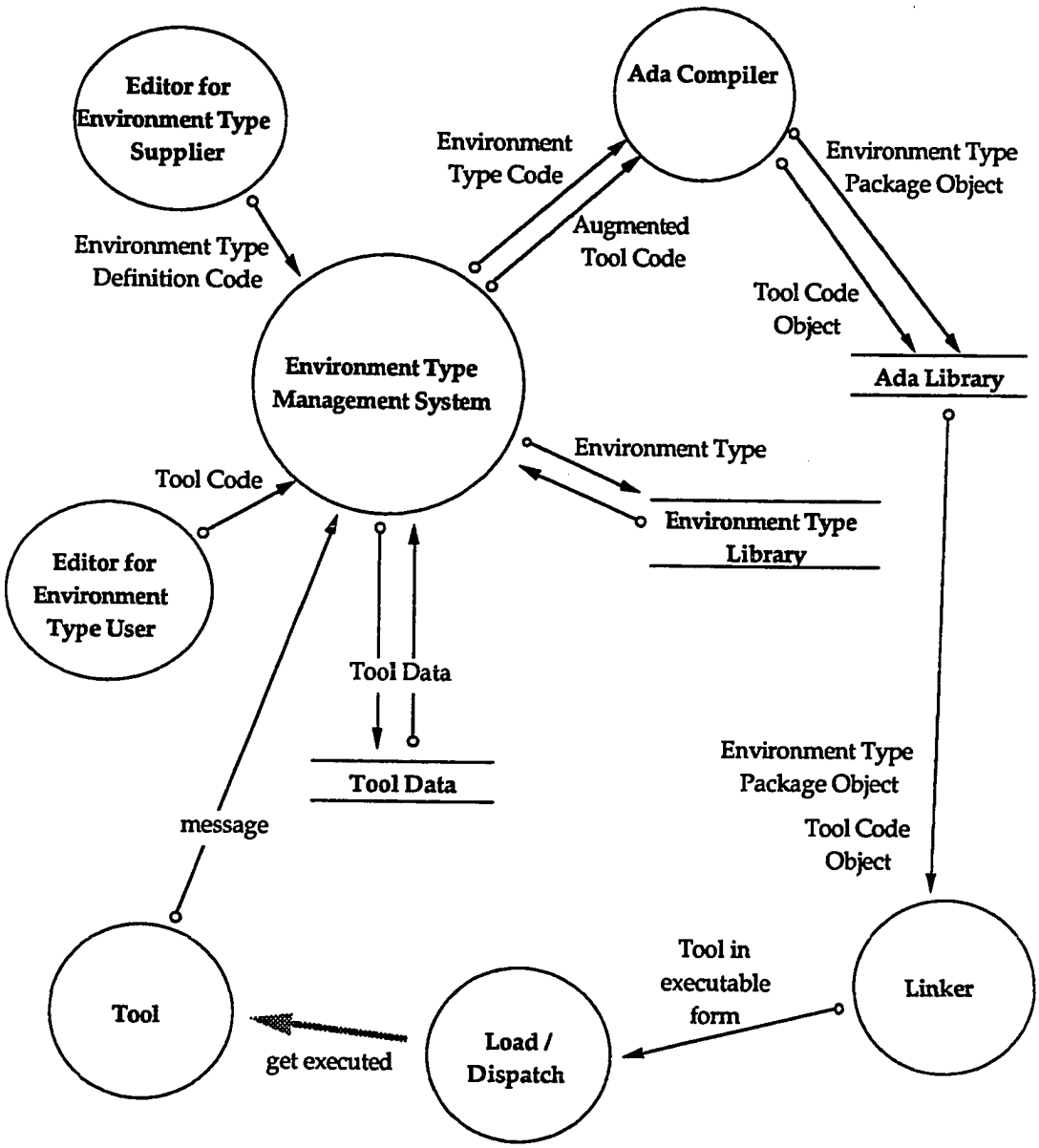


Figure 5.3. Data Flow Diagram for Type System

from the environment type definition code. The environment type code is, in this research, an Ada package per each environment type.

The environment type code is later compiled and catalogued into the Ada library. These object file images in the Ada library are used when the tool code references the environment type. The operational definitions for the environment types are translated into Ada code and encapsulated in package structure by the environment type management system.

The tool code is written using the host language (Ada) using necessary references to environment types and instantiation of tool data, and applying the appropriate operations on them. Any properties of the environment type use are checked statically by the environment type management system. After the tool code is checked for validity of type use, the environment type management system generates the augmented tool code which is host language compatible. This code contains other codes to manipulate tool data appropriately, that is, search objects, send message, manipulate data structure for tool data, and so on.

The augmented tool code is later compiled and linked to generate an executable form of the tool. During runtime of the tool, the actual creation, use, and manipulation of the tool data are executed. Any reference to tool data is managed by the environment type management system. The tool sends messages to the tool data. The actual message traffic is controlled by the environment type management system to decode message and effect tool data status change.

The environment type management system consists of four component systems:

1) the environment type definition processor (ETDP),

2) the tool compiler,

3) the environment type library manager (ETLM), and

4) the tool data manager (TDM), as shown in Figure 5.4.

The ETDP takes the environment type definition code and generates translated environment type code. It requests the ETLM to query existing environment types referenced for the definition and catalogue new environment types. The ETDP consists of the ETDP Lexical Analyzer/Syntax Analyzer and the ETDP Code Generator. The ETDP Lexical Analyzer/Syntax Analyzer processes input environment type definition code to check the validity of any references to existing environment types and generates a symbol table and a syntax tree. The ETDP Code Generator generates the environment type code written in the host language. The environment type code consists of the specification part of the environment type and the operation definition part of behavior for the environment type. The specification part becomes Ada package specification and the operational part becomes Ada package body. When the environment type definition references any environment types – for instance, any use of specialization, include clause, or content type defined as environment type – the referenced environment types are queried through the services of the ETLM.

The tool compiler scans tool code to check the validity of environment type uses by querying the environment type. The validity of use of name, operation, and parameter for the environment type is checked in this phase. The tool code is translated to augmented tool code, which is written in the host language. The augmented tool code includes the code to generate data structures for tool data, send messages to tool data, and manipulate tool data.

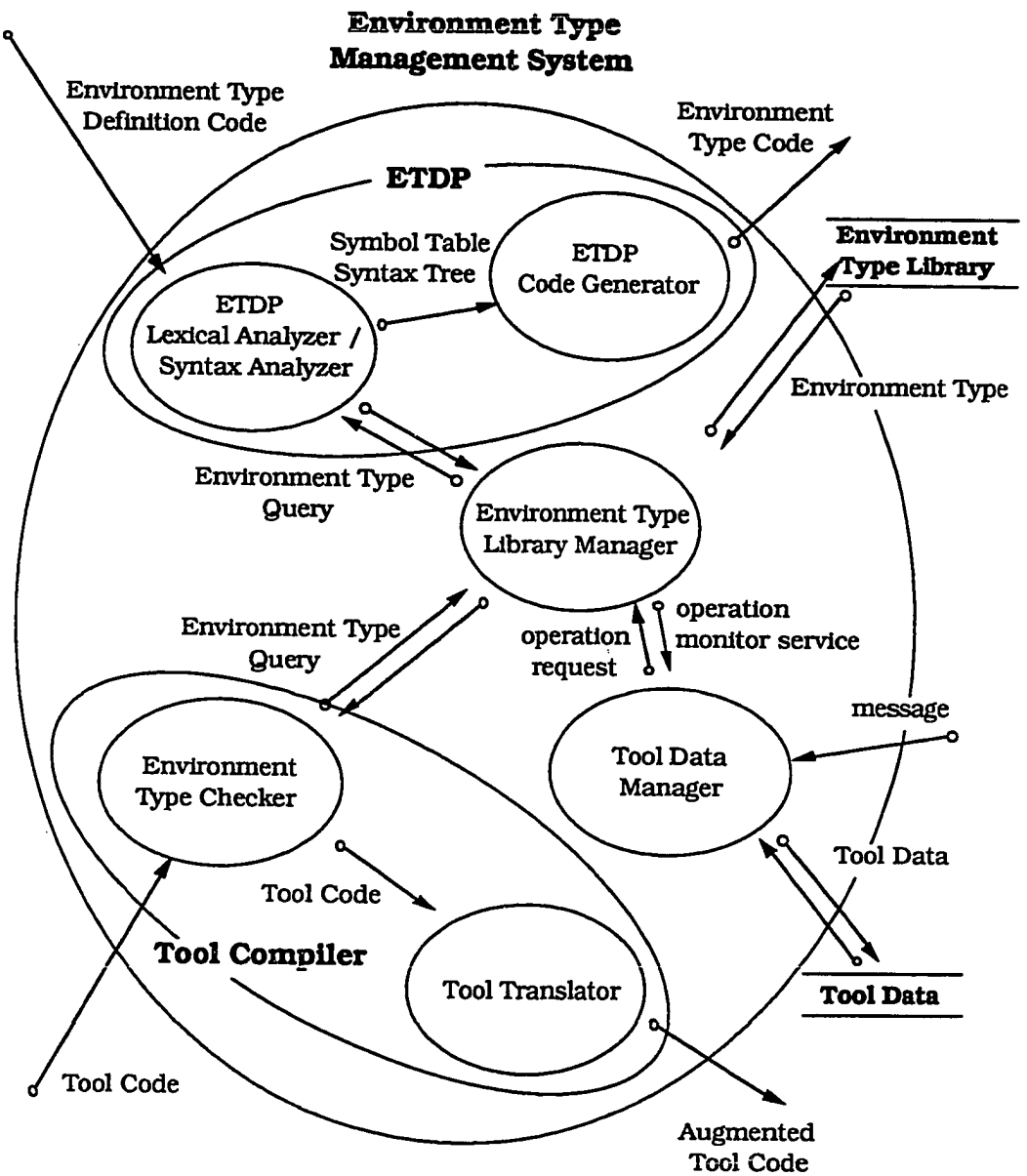


Figure 5.4. Detailed Data Flow of Environment Type Management System

The ETLM is a concurrent entity used to provide the services of querying and cataloguing the environment type. The environment types are strictly managed only by the ETLM to support the integrity of environment types. The ETLM catalogues the environment types by creating appropriate data structures into the environment type library. The environment type structure includes information about contents, attributes, relations, and operations. The set of operations for each environment type is constructed as a monitor process to provide necessary mutual exclusion in manipulating tool data. Upon request of operation for the environment type, the ETLM searches the monitor process for the environment type and returns it to the requestor.

The TDM constantly receives and decodes messages sent from the tool to arrange appropriate monitor services through ETLM. The message contains the following information:

1) environment type name,
2) operation name,
3) parameter for the operation, and
4) tool data that the operation is applied on.

The TDM requests the ETLM to search the monitor process for the environment type and arranges for the monitor process to run. When the monitor process runs, the operation code is executed to manipulate the tool data.

5.2.3 Structure Chart

The structure chart in Figure 5.5 shows the component subsystems in the environment type management system. The control accesses and the data flows among subsystems are designated.

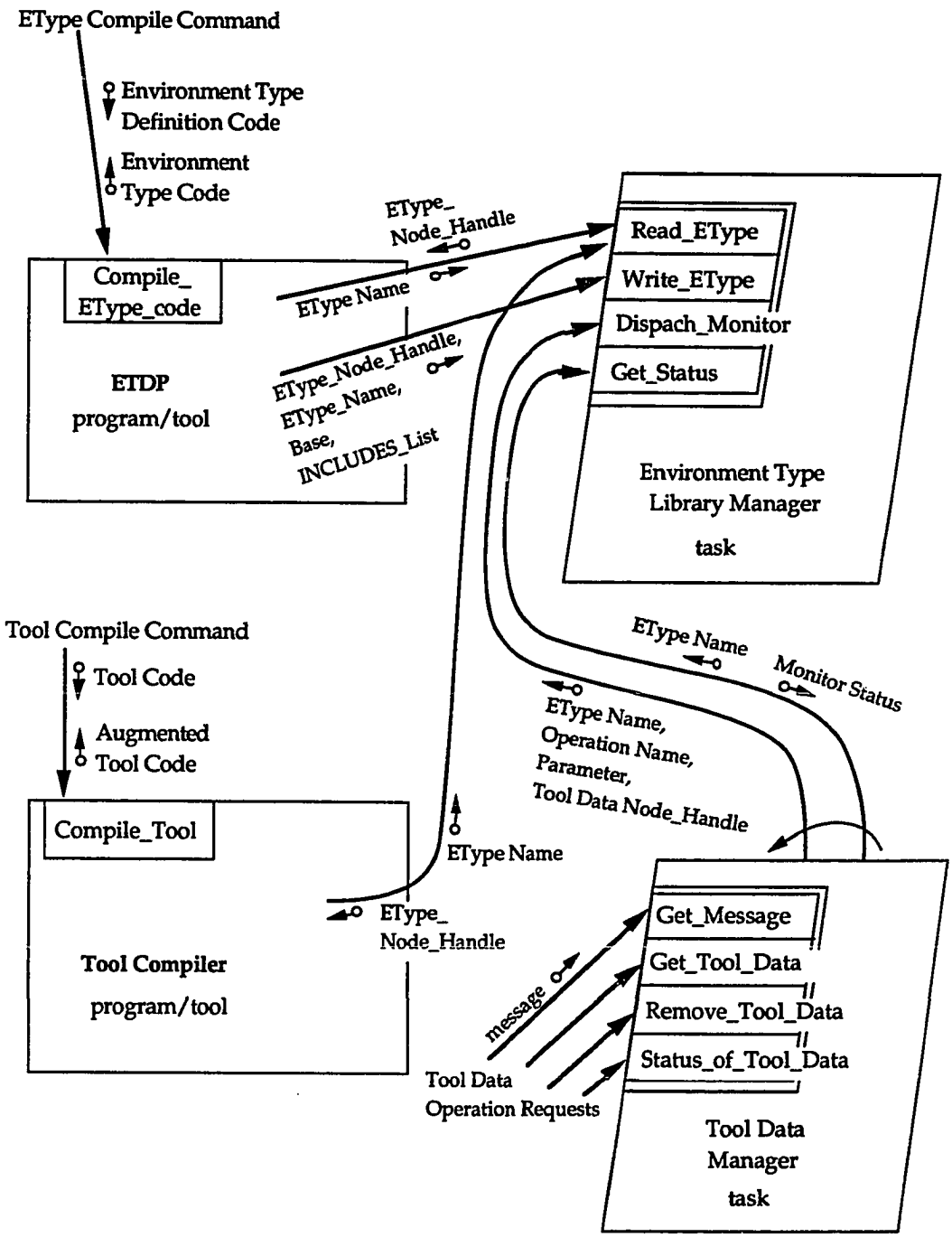


Figure 5.5. Structure Chart of the Environment Type Management System

The ETDP and tool compiler are defined as programs (separate tool) or packages. The ETLM and TDM are defined as concurrent entities (task). Interfaces for each subsystem and control accesses among subsystems are defined. The ETDP takes environment type definition code from the environment type compile command and outputs environment type code. The tool compiler takes tool code and outputs augmented tool code. It calls Read_EType to query the environment type.

The ETLM has entries of Read_EType, Write_EType, Dispach_ Monitor, and Get_Status. These entries are enclosed in a *select* construct to take only one entry call at a time; this is because the environment type library managed by these entries is shared among various service requests. The Read_EType finds the environment type and returns the node handle for the environment type structure.

Write_EType catalogues new environment type into the environment type library. The Dispach_Monitor searches the monitor process for the environment type to manipulate the tool data. The Get_Status returns the status of the monitor process of the environment type.

The TDM picks up the messages of tool data operation requests. Get_Message picks up the next available message from the message queue and requests Dispach_Monitor of ETLM. Get_Tool_Data constructs a node structure for the tool data of a given environment type. Remove_Tool_Data closes the access of the tool data from the tool. Status_of_Tool_Data requests Get_Status of ETLM to find the status of the monitor process of the environment type.

5.3 Environment Type Definition Processor (ETDP)

The ETDP analyzes the environment type definition code to create new environment types in the environment type library. These environment types are constructed in compliance with the CAIS node model and provide all information about the type. The environment type definition language supports full functions of the object-oriented mechanisms including type evolution and polymorphism.

The ETDP consists of several subsystems; Lexer, Parser, EType Checker, and ETDP code generator, and uses the Ada tools of Alex and Ayacc. Alex is a lexical analyzer that inputs the regular expression of identifiers of the environment type definition language. The output of Alex is a DFA (Deterministic Finite State Automata) table. Ayacc is a compiler-compiler to input syntax of the environment type definition language and output an LALR parsing table.

Lexer takes the DFA table to process environment type definition code. The output of Lexer is a symbol table and a token stream of given environment type definition code. Parser takes LALR parsing table and analyzes the syntactic information and returns a symbol table and a syntax tree. EType checker inputs the symbol table and syntax tree and calls Read_EType for any reference of the existing environment types. The resulting symbol table and syntax tree of the EType checker is used in ETDP code generator. The ETDP builds a node structure of the environment type structure and calls Write_EType to catalogue it into the environment type library. It uses EType_Name, EType_Node_Handle (to point to an environment type node structure), Base (to point to a supertype node), and a list of included environment type names to specify the location of the node

structure when it is catalogued in the library. Figure 5.6 shows the structure chart of the ETDP.

The operational definition part of the environment type definition code is translated to an Ada package body named after the environment type name. The code for the package body is Ada code used to define operations that are Ada procedures or functions.

5.4 Tool Compiler

The tool code written in the host language (Ada) references to environment types by commands described in Chapter 4. The reference to the environment type or the use of tool data can be any of three cases:

1) defining new tool data,
2) sending a message to the tool data, and
3) closing use of the tool data.

The tool compiler scans the input tool code to encounter any code referencing the environment type or using tool data. Necessary type checking must be performed, and the translated host language that manipulate tool data must be generated and inserted into output code. Figure 5.7 shows the subsystems of the tool compiler.

5.4.1 Environment Type Checker

Before the translation of commands manipulating tool data into the host language, any references to environment types in tool code must be checked in a static way. Any references to the tool data must be checked at runtime of the tool since the tool data are runtime objects.

The operations on the tool data are translated to the codes to provide message-sending through an interprocess communication channel. Actual messages sent to tool data are picked up by the TDM during runtime of the

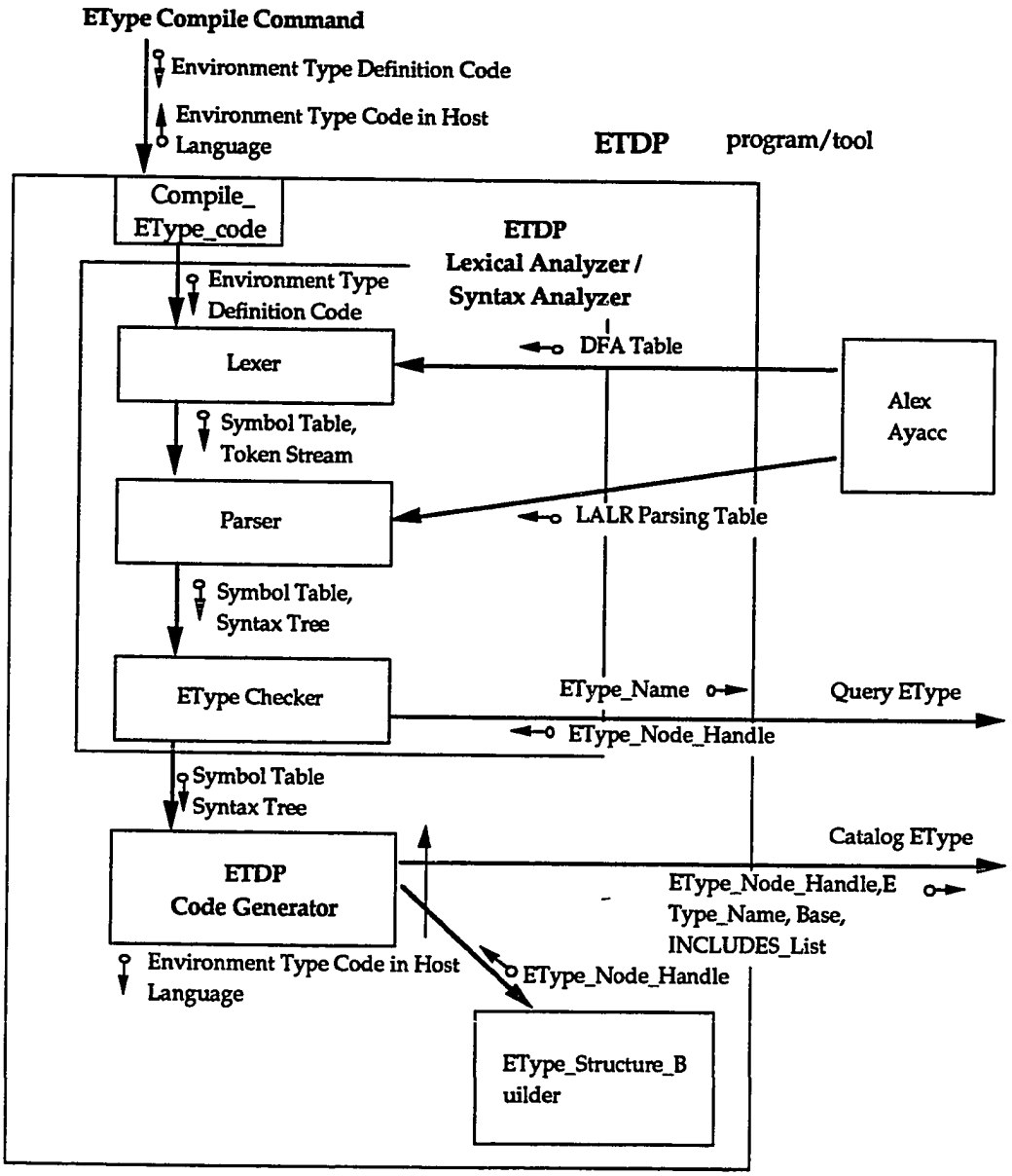


Figure 5.6. Structure Chart of the ETDP

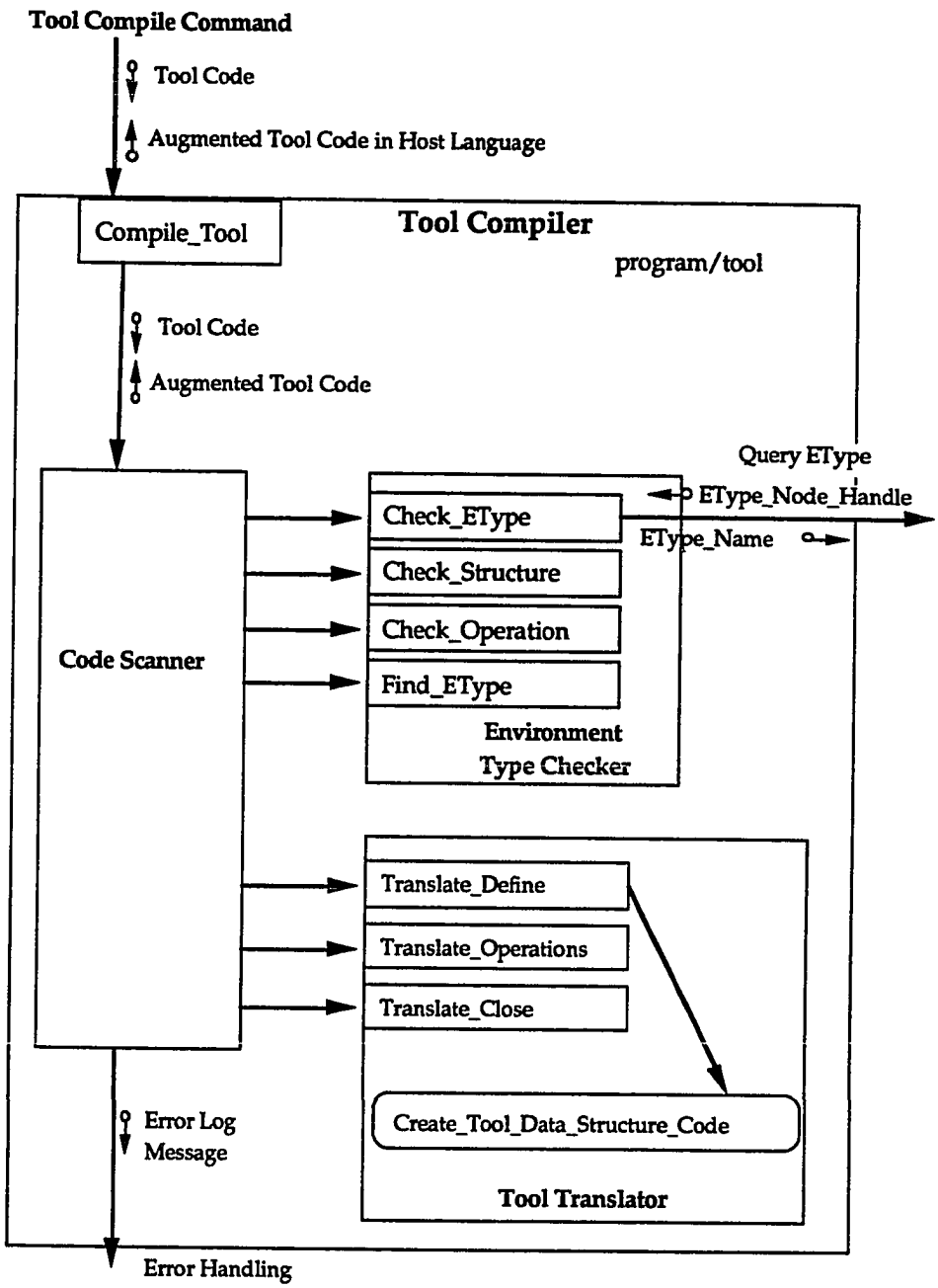


Figure 5.7. Structure Chart of Tool Compiler

tool. The tool compiler must check the validity of the use of tool data including the existence of named tool data, operation names, parameter names, and parameter types. After the necessary type checking is completed, the tool compiler generates codes to manipulate tool data.

The type checking facilities can be defined in the package of Environment Type Checker, as shown in Figure 5.7. The tool compiler code scanner calls Environment Type Checker to query necessary information for applying environment types.

5.4.2 Translation of Tool Code

The ***includes*** statement enables the referenced environment types visible in the tool code. The includes statement is translated into the Ada ***with*** statement with a corresponding package name for the environment type. The package name can be found in the Ada library, where all of the defined environment types are registered as packages, to provide operational definitions of the environment type from ETDP.

```
includes WINDOW;          can be translated as

with WINDOW;
```

The ***define*** statement makes the tool data accessible from the tool. If there is no instance object of the given environment type (the reference of the environment type is new) in a given pathname, then a new tool data structure is created. Otherwise, existing tool data of the given pathname are opened under the given intent. After define, the tool can reference the tool data.

The tool compiler performs the necessary type checking for the reference to environment type and generates code to make the tool data structure. To create this structure (store for the environment type), the tool

compiler generates code to call Get_Tool_Data of the TDM. The call is a task entry. When the TDM accepts Get_Tool_Data, the actual structure of the tool data is constructed. Necessary translation includes Etype_Name, Pathname, Intent, and Tool_Data_Node_Handle. The define command

```
define my_window:WINDOW("'USER(JONES)'WORK",WRITE);
```

can be translated as task entry call

```
TDM.Get_Tool_Data
        (my_window,"WINDOW","'USER(JONES)'WORK",WRITE);
```

The node handle my_window is a local identifier in the tool code that references the tool data structure, which is global and persistent to all relevant tools. Other tools may open the same tool with the same pathname but with other name for the node handle. The actual references to the tool data, however, are directed to the same node structure since tools share the same pathname.

The purpose of the above define statement can be interpreted as follows: the tool opens tool data of the abstract environment type WINDOW in the pathname of WORK directory of user JONES to write some data into it. The actual translation can be performed only after checking the environment type WINDOW.

The ***send*** statement is used to apply operations to the tool data. Send takes the operation name with corresponding parameters for a defined tool data name. The tool compiler performs type checking to verify the validity of the operation and parameter types for environment type of the tool data. For example, environment type WINDOW has an operation of

```
operations
   procedure scroll (lines:in integer);
```

The tool code defines tool data my_window as

```
define my_window:WINDOW;
```

and applies the operation as

```
send my_window.scroll(5);
```

The tool compiler checks the validity of the operation name "scroll" for the environment type of the tool data "my_window". Then, the tool compiler checks the validity of parameter "5" against the formal parameter of operation "scroll", which is integer. The result of the operation "scroll" will be applied on the tool data structure identified as node handle "my_window". The resulting translation will be a code to write a message to the message queue using CAIS queue_io. The tool data node handle, environment type name, operation name, actual parameter, and kind of operation are parameters of the queue_io's write interface.

```
queue_io.put(my_window,"WINDOW","scroll","5","procedure");
```

During runtime of the tool, the message will be written into the queue file; the TDM will pick up the message and decode it to allow the monitor process to execute the operation "scroll" defined in the environment type "WINDOW".

If the operation is defined in the inherited environment type, then the tool compiler must find the operation from the appropriate environment type and translate the target tool data structure accordingly. For example, if the environment type "WINDOW" is a specialized type of "TEXT_FORM" and one of the operations, "APPEND_CHARACTER", is inherited to "WINDOW", then tool code using "WINDOW" can manipulate tool data of type "WINDOW" with "APPEND_CHARACTER". The tool compiler, in such a case, must search for the operation "APPEND_CHARACTER" in the

environment type structure to decide the correct data structure to manipulate. Once the environment type for the operation is found, the operation should be applied to the corresponding type's stores in the tool data structure.

```
environment type TEXT_FORM
.....
   operations
        .....
        procedure APPEND_CHARACTER (C:in character);
end TEXT_FORM;

environment type WINDOW specializes TEXT_FORM
.....
end WINDOW;

-- tool code includes WINDOW
   define my_window: WINDOW;
.....
   send my_window.APPEND_CHARACTER ("A");
```

In the above example, the tool compiler decides where the operation "APPEND_CHARACTER" must be applied in the node structure of the tool data "my_window". The correct place of the operation applied is the specialized area of the tool data "my_window" since the operation is defined in supertype "TEXT_FORM". The details of the node structure for both environment type and tool data are described in the following sections.

When an operation of the send statement is a function or includes out mode parameter, the tool compiler must prepare the appropriate store for receiving the result of the function. The type compatible store must be set up so that augmented tool code can use it.

The ***close*** statement is used to declare the end of tool data use. Once the tool calls close, it cannot apply the send operation to the tool data without another define statement. Close takes a tool data node handle and can be translated as a CAIS close statement for this node handle. Even though one tool closes the tool data, other tools or other parts of the same tool may still reference the tool data. The actual closing of the tool data (such as deallocation of tool data structure) will be postponed until all uses of the tool data are closed. The TDM handles actual deallocation of the tool data structure dynamically upon request of close to remove the tool data.

5.4.3 Operation Dispatching

Operations for the environment type are defined in the environment type definition. The data structure (value store) of the environment type for defined tool data is constructed for each define statement. In other words, one environment type can have multiple instance objects of tool data. Each tool data has a different value status even though their structures are identical for the same environment type.

Operations defined in the environment type are used to manipulate the value store of tool data. As an autonomous entity, the tool data have a concurrent process called a monitor process to manage the operations defined for each environment type. The monitor process controls scheduling of the operations for all instance objects of tool data for each environment type. The monitor process is attached to each environment type structure since the

codes of the operations are shared resources among all tool data of their environment type. Only one of the operations can be allowed to execute at a time.

The tool data are autonomous entities of abstract data. They must be available to all relevant requests for the query of its encapsulated data. All communication between the tool (requestor) and tool data (requestee) is handled through the message. From the viewpoint of tool code, the tool sends a message to the tool data, which receives the message and decodes it to apply the requested operation to the data. However, in a physical schema, the message communication is handled through the message queue management. When tool code contains the code to send a message to the tool data, the tool compiler translates it to the proper code for writing it to the message queue. The message written to the queue is picked up by the TDM, which is constantly receiving the next available message from the message queue. The message is decoded and causes the monitor process to execute the operation to manipulate the tool data value. The mechanism of message management is described in Figure 5.8.

The message queue can be implemented as a solo_queue of CAIS that resembles a ring buffer. The message is selected on the basis of first-come-first-serve.

5.5 Environment Type Library Manager (ETLM)

Environment types are stored in a central repository called the environment type library. The environment types are reusable objects to construct another environment type by user during execution of software engineering environment system. The environment types catalogued in the environment type library are managed by the ETLM, which handles queries and

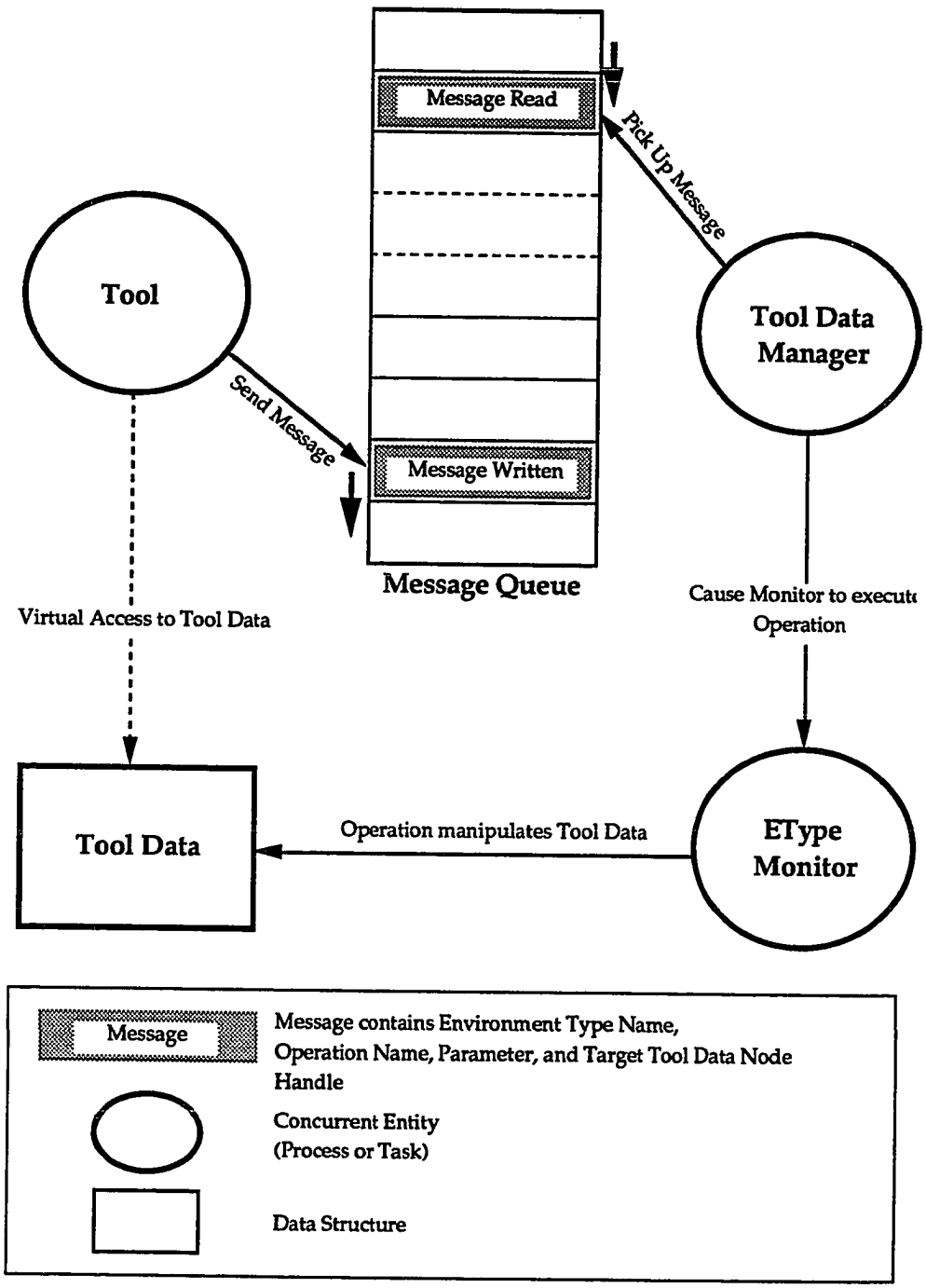


Figure 5.8. Message Management among Tools and Tool Data

catalogues new environment types. Operations defined for the environment types are stored as shared resources to manipulate tool data for corresponding environment types. The ETLM schedules a monitor process that manages operations of the environment type instances.

The ETLM has four basic services: Read_EType, Write_EType, Dispatch_Monitor, and Get_Status. Interfaces for the ETLM are described in Figure 5.9. The ETLM is an autonomous entity (defined as a task) that provides services to various sources of requests from other programs or tools running in the environment type management system. The interfaces of the ETLM share the environment type library so that they are enclosed by the *select* clause of the task.

Read_EType searches the environment type library for the environment type structure using the environment type name (EType_Name) and returns a node handle for the environment type structure (EType_Node_Handle). This interface can be used to find the existence of the named environment type or details of the environment type structure. The search will navigate relationships from the node BOTTOM, which is used to collect every environment type defined in the environment type library. The environment types have unique environment type names that are used as keys of the primary relationship "E_TYPE_DEF".

The environment type lattice is shown in Figure 5.10. The node TOP is the root of all environment types defined. The node for each environment type has a secondary relationship, "specializes_to", that emanates from the supertype node pointing to it. The inheritance structure using the relationship of "specializes_to" constructs a general type lattice structure. The secondary relationship "includes" provides visibility to search any inherited

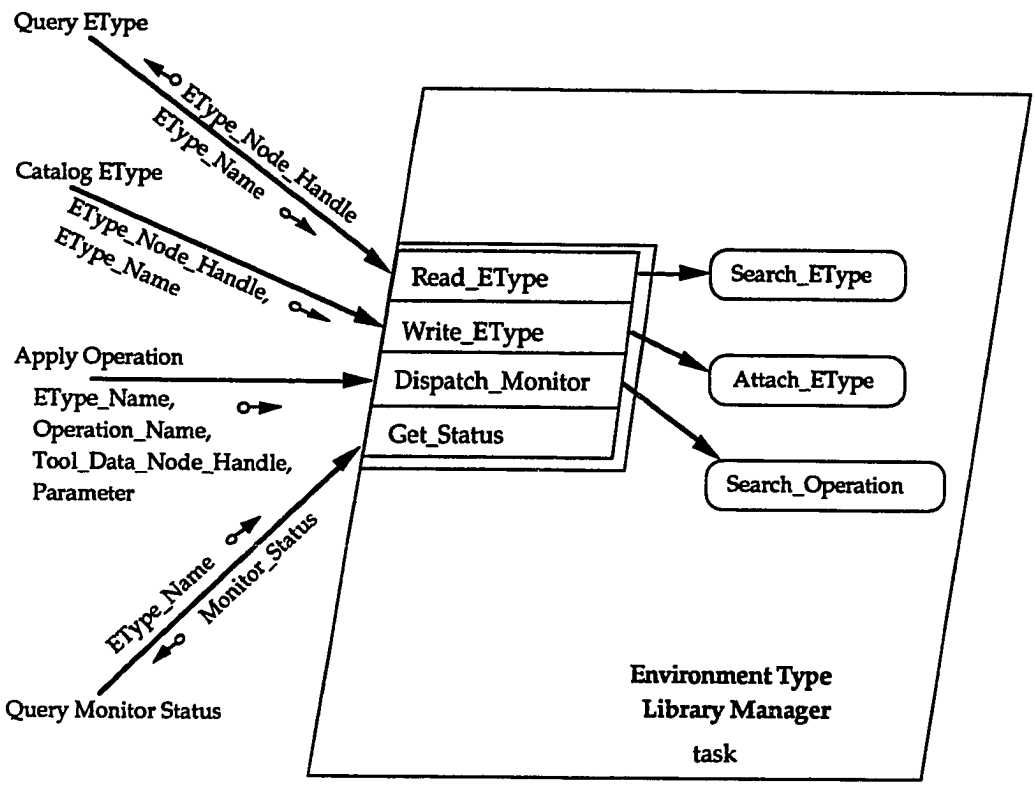


Figure 5.9. Structure Chart for ETLM

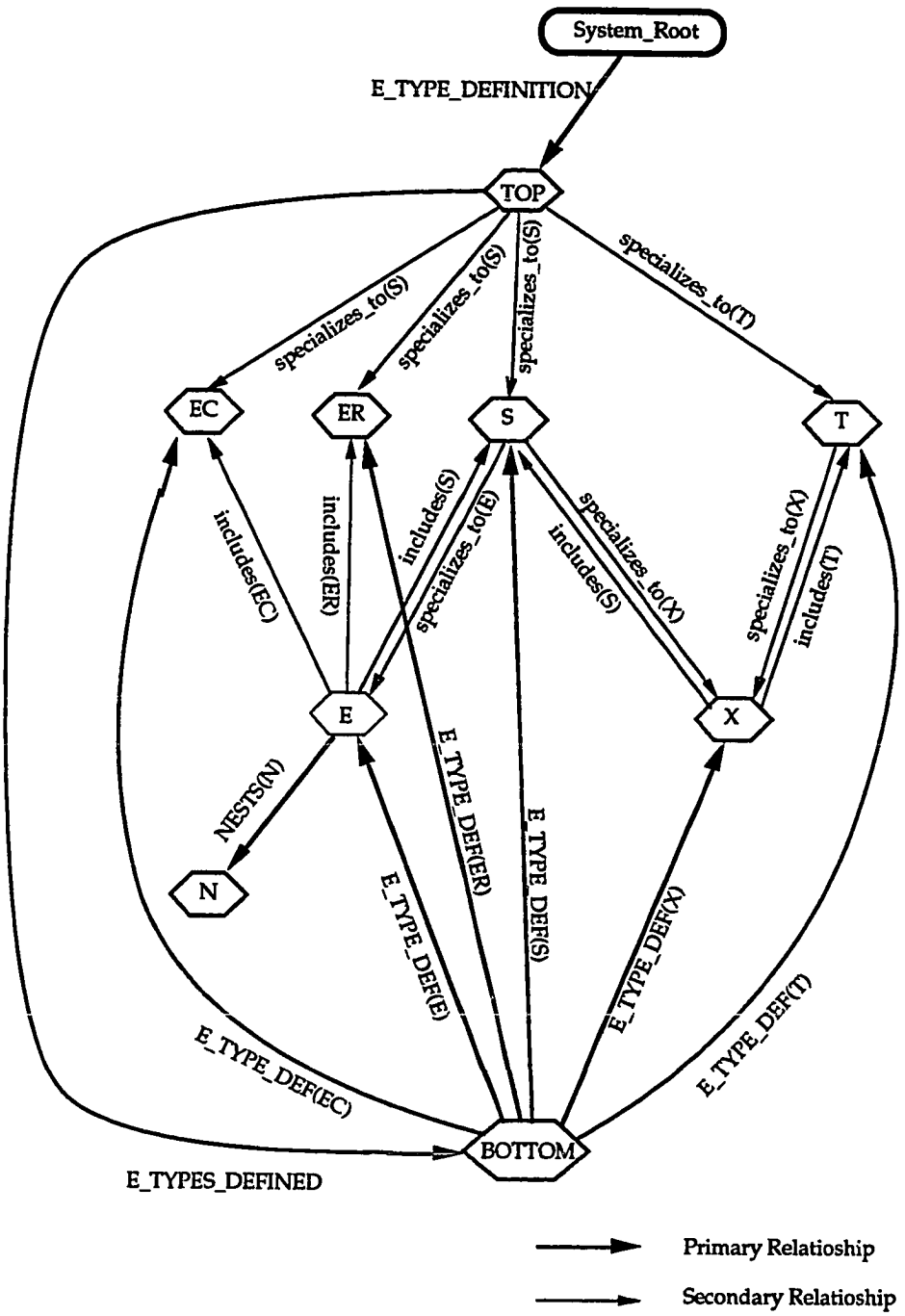


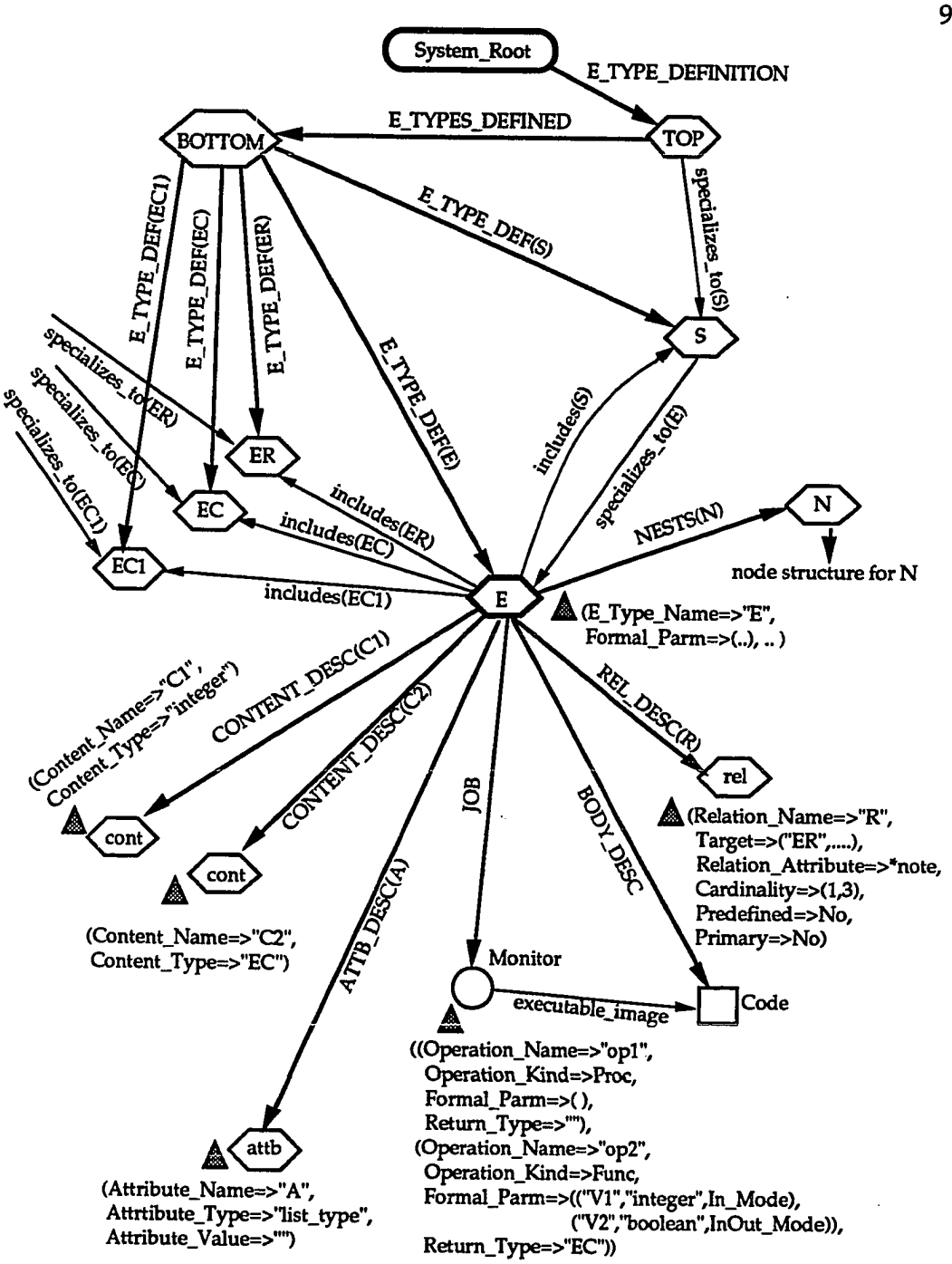
Figure 5.10. Environment Type Lattice

properties for the environment types. For example, the environment type "E" in Figure 5.10 is specialized from "S" and uses "EC", "ER", and "S" for definition. The environment type "E" inherits every property for "S" as designated by the relationship "specializes_to(E)". The environment type "E" uses the "EC" to define some of the content field of it. So, the properties of the "EC" are inherited to the content of "E". The environment type "ER" is used to define the relation of the target for "E". No property of "ER" is inherited to "E".

Multiple inheritance can be achieved by taking more than one "specializes_to" relationship as "X". Every environment type node has the primary relationship "E_TYPE_DEF" emanating from node BOTTOM to it. To search the named environment type structure, the ETLM navigates relationships "E_TYPE_DEF" from node BOTTOM. The environment type library is constructed to resemble the type lattice structure. The representation of the type lattice structure can be an entity-relation model using the CAIS node model, as shown in Figure 5.10.

Write_EType catalogues a new environment type to the environment type library. Information defined in the environment type definition code is interpreted to construct a node structure as in Figure 5.11.

The environment type structure is constructed around the structural node (environment type node), which has a primary relationship "E_TYPE_DEF" with a key using the name of an environment type emanating from node BOTTOM. The secondary relationships "specializes_to" emanates from the supertype environment type node if the environment type is a subtype. All environment types used for definition, as



*note : Relation_Attribute has identical attributes as in the attribute definition.

Figure 5.11. Node Structure for Environment Type Definition

in the includes statement, are linked by secondary relationship "includes" that emanates from the new environment type.

The contents of the environment type are built as structural nodes for "cont" using primary relationship "CONTENT_DESC" with the key using content field names. The name and type of the contents are stored as CAIS attributes of the content nodes. If the content type is generic, the actual type name bound in run-time will be assigned by an actual type parameter when the tool data are instantiated.

Attribute fields of the environment type are constructed as attribute nodes, "attb", using the primary relationship "ATTB_DESC" with key using attribute names. Attribute name, type, and initial value are stored as CAIS attributes of the attribute nodes.

Relation fields are constructed as relation node, "rel", using the primary relationship "REL_DESC" with key using relation name. The name of the relation, name of the target environment type, relation attributes, cardinality, and other information are stored as CAIS attributes of the relation node.

The set of operations for the environment type is structured as a process node to make the monitor process node, "monitor". The information for each operation such as name, kind, and formal parameters of the operations are stored using CAIS attributes for the monitor process node. The executable image of the monitor process is stored in the file node "code" using the primary relationship "BODY_DESC".

The nested environment types are structured using the same node model and are attached to the environment type node by the primary relationship "NESTS" with key of nested environment type name. The

nested environment types are not visible outside the definition of new environment type. Therefore, the nested environment types are not linked to node BOTTOM.

Dispatch_Monitor searches for the operation of the tool data of environment type name and dispatch the monitor process stored in the environment type structure. The search takes place as follows:

1) find the environment type name for the tool data,
2) find the operation name from monitor process node of the environment type structure,
3) if not found in step 2), find in the nested environment type,
4) if not found in step 3), find in the environment type of the content inherited,
5) if not found in step 4), find the supertype of the environment type and repeats steps 2) through 5).

After the search, the name of the environment type defining the operation is returned and the monitor process node is set to run in the next process scheduling. The monitor process will decode the message to get the name of the operation and parameters and apply the operation to the tool data. The environment type structure found in the operation search gives information about which part of the tool data structure is the target of the operation application.

Get_Status finds the status of the monitor process for the named environment type. The status can be defined as the status of the CAIS process. This interface is used to query the status of the monitor process by tools.

5.6 Tool Data Manager (TDM)

Tool code written in the host language (Ada) includes commands to manage the tool data such as defining a new tool data structure, closing the tool data, sending messages to the tool data, and querying the status of the monitor process. These commands are translated by the tool compiler to generate necessary actions during runtime of the tool. The action taken during runtime requests the appropriate services from the TDM, which include: Get_Tool_Data, Get_Message, Remove_Tool_Data, and Status_of_Tool_Data. These interfaces are shown in Figure 5.12. The TDM is an autonomous entity such as an Ada task or process to service various requests from tools in runtime.

Get_Tool_Data is requested from the define command. It constructs a new tool data structure under a given pathname. To build a node structure for the tool data, the TDM constructs all the stores for a designated environment type as well as all inherited environment types. The stores for the environment type used in content fields and NESTed types are also necessary. Although some of the supertypes can be shared among many environment types, actual tool data must carry individual stores for all data including a store for the inherited types. Based on such a concept, the tool data structure has been designed as in Figure 5.13.

The tool data structure consists of tool data node, stores for the environment type defined, stores for the NESTed environment types, and stores for the specialized environment types. The store for the specialized environment type recursively links stores for its supertype. If content is defined as an environment type, the store for the type is constructed in the same way. In Figure 5.13, the shaded structural nodes are stores for the

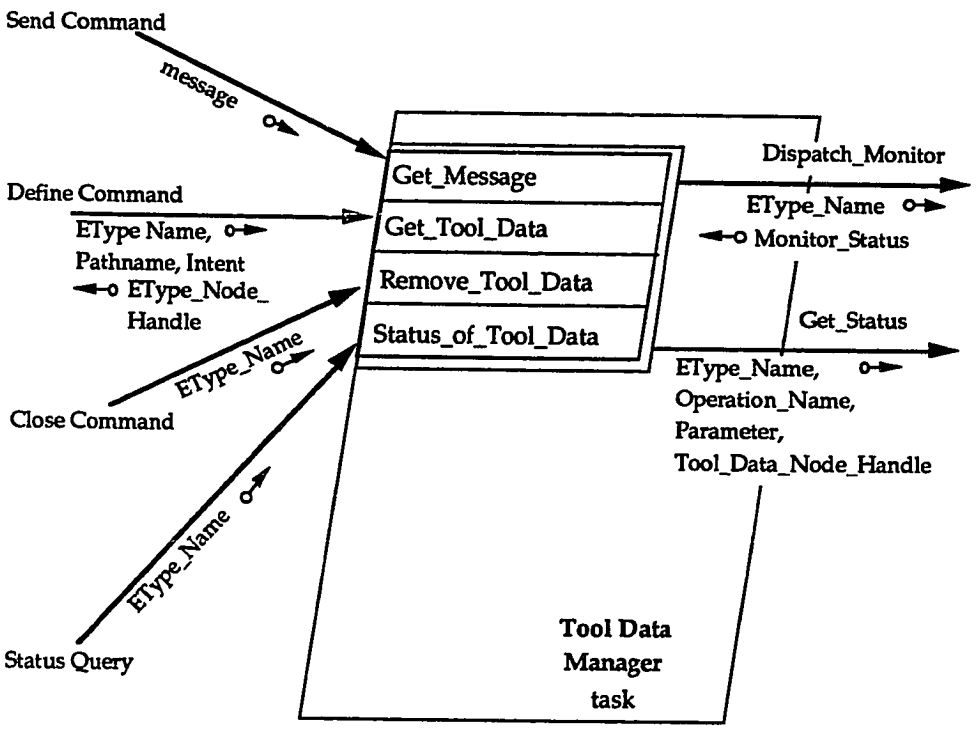


Figure 5.12. Structure Chart for TDM

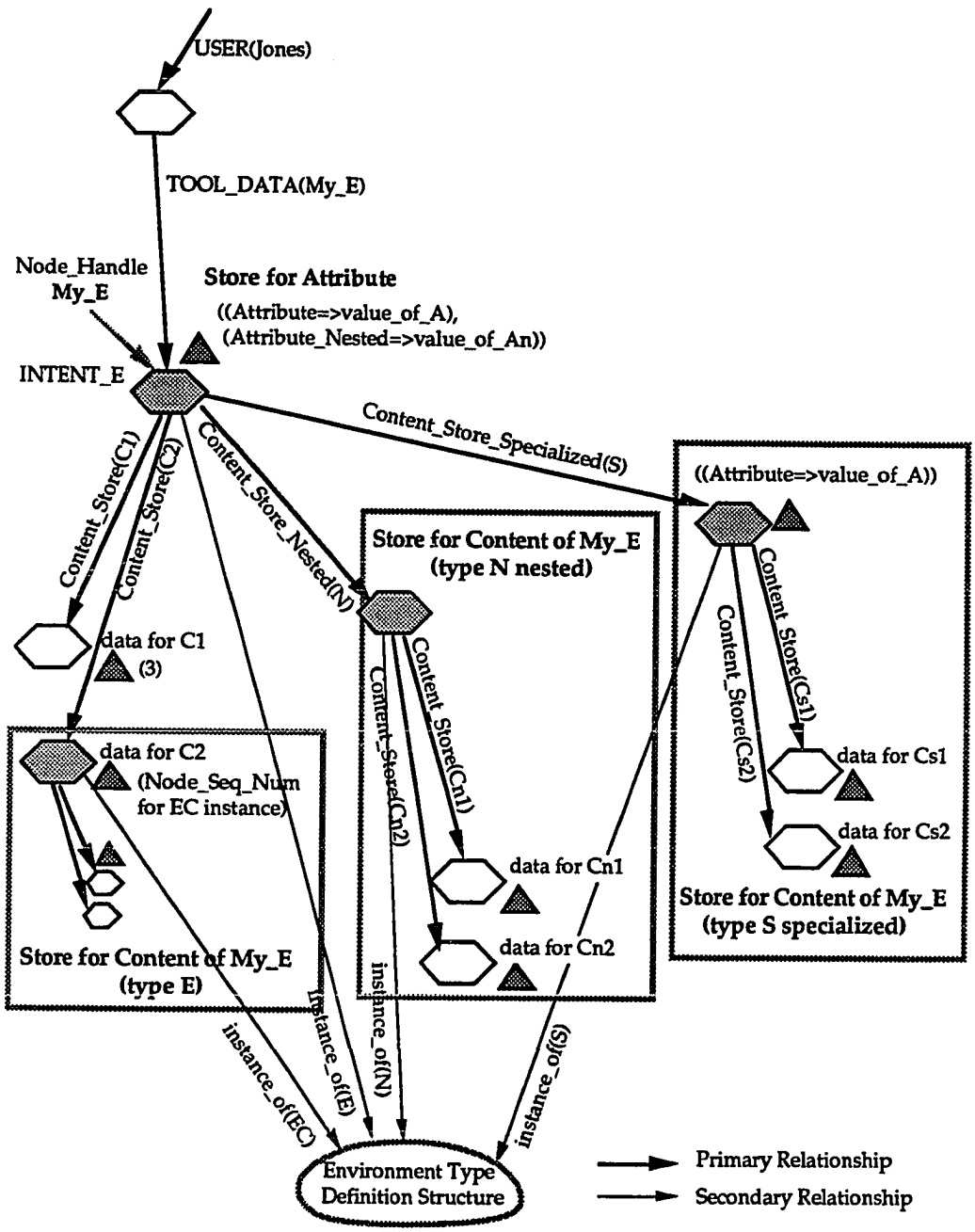


Figure 5.13. Design of Node Structure for Tool Data

environment types while the white structural nodes represent stores for the regular Ada type.

All structural nodes representing stores for the environment types have a secondary relationship, "instance_of", to the node-defining environment type used. The primary relationship "CONTENT_STORE" links content nodes representing content fields of the environment type. The primary relationship "CONTENT_STORE_NESTED" links structural nodes representing stores for the NESTed environment type used. The primary relationship "CONTENT_STORE_SPECIALIZED" links structural nodes representing store for the specialized environment type node.

Attributes of the environment type are stored as CAIS attributes for tool data node. Attributes for the inherited environment types (type of content field, NESTS, and specialized environment type) are stored in their nodes individually. When new tool data are instantiated, their structure is constructed and their primary relationship "TOOL_DATA" is set from a node named as the pathname. The intent of the tool data node is set to the given intent parameter. After all construction of the tool data component is finished, the node handle pointing to tool data node is returned.

To create a tool data structure, the tool compiler references the environment type structure through the services to ETLM (interface of Read_EType). The action includes necessary CAIS calls (such as CREATE_NODE) to make each data store for the content and attribute. For example;

```
define my_window: WINDOW("'USER(JONES)'WORK",WRITE);
```

can result in the following steps:

1) create a new tool data node in the given pathname with the given intent,
2) create the data store for the content fields of the environment type WINDOW,
3) create the data store for the attribute fields of the environment type WINDOW,
4) create the tool data node structure (including tool data node and data store for content fields and attribute fields) for any nested environment type of WINDOW,
5) create the tool data node structure recursively for specialized environment types (supertypes) of WINDOW, and
6) return the node handle for the tool data node.

If a tool data node structure is present, then the code generated will be that to open the tool data node with the given intent. During the runtime of the tool, the success of the node open for the tool data depends on the access mechanism of CAIS. CAIS provides a necessary locking mechanism for node open. The interfaces of the TDM are enclosed in a *select* construct of the task.

Get_Message picks up new messages from the message queue. This is done by CAIS QUEUE_IO. The names of the environment type of the tool data and the target tool data node handle are decoded, and the TDM requests service of Dispatch_Monitor of ETLM.

Remove_Tool_Data closes the node handle for the tool data. The last request the Remove_Tool_Data deallocates the node structures of the tool data.

Status_of_Tool_Data requests Monitor_Status of the ETLM for the given environment type name of the tool data.

6. ANALYSIS OF THE TYPED SOFTWARE ENGINEERING ENVIRONMENT

A typing mechanism for the software engineering environment has great potential for improving the quality and productivity of software development. Dynamic evolution of environment types are important to support user definable types during runtime. The software engineering environment can be most useful when it is equipped with such a type system. However, implementing such a system is not an easy task. This chapter examines the considerations necessary to implement the type system for the software engineering environment.

The analysis in this chapter has been done for several reasons: to identify procedural factors of software development in a typed software engineering environment, to assess the pros and cons for using typing mechanisms in a software engineering environment, to find feasibility and compatibility in the implementation of a type system for a software engineering environment, and to estimate potential contributions to the software development using a typing mechanism.

The method taken in this analysis is based on qualitative information because of a lack of experimental data on working systems that use the typed software engineering environment. Although the analysis is based on informal knowledge, it is believed that the typing mechanism has a great potential to contribute to software engineering both in theory and practice.

The categories of the analysis consist of

1) a use analysis of the typed software engineering environment from the user's viewpoint,

2) an implementation analysis of the typed software engineering environment from the developer's viewpoint,
3) quality improvement aspects, and
4) productivity improvement aspects.

6.1 Use Analysis : User's View

Once the typed software engineering environment is established and a proper system is installed to support such an environment, software development roles (project management role, configuration management role, program development role, environment adoption role, and so on) are affected by the dynamic typing mechanism of software development. Programs such as tools or applications are composed of tool data using environment types, which evolve dynamically. The evolution of environment types includes cataloguing new types, specializing existing types, and deleting obsolete types.

Such a new paradigm can change the existing practice of software development. As the members of a software development project use such typing facilities for their own needs, there are several possible impacts. From the viewpoint of users who create/supply new environment types as well as those who build tools or applications, there are several useful guidelines.

6.1.1 Guidelines

6.1.1.1 Guidelines for the Environment Type Creator

The environment type creator designs and implements new environment types and catalogues them into the environment type library. As the environment type management system supports dynamic environment type creation, user-definable environment types can be created in the system dynamically while the predefined environment types can be created in static manner.

The environment types are important objects shared by many tools. New environment types must be designed well enough to be standard yet be long-lasting in a software environment. The environment type creator must meet the following criteria when designing new environment types:

1) Environment types must be stable.

Environment types are used to instantiate tool data or to define other environment types. The tool data carrying information among tools are the abstract data in environment system. Once catalogued, the modification of the environment types is not desirable because changes to existing environment types can cause undesirable side-effects to instantiated tool data. Therefore, the environment type creator must build new environment types with extreme care, which requires detailed analysis of and experience with such environment types.

2) Environment types require good testing.

Once a new environment type is designed and implemented, thorough testing is required prior to cataloguing the environment type into the environment type library. Necessary documentation regarding the new environment type must be attached to provide guidelines for use, exceptions, and descriptions of the interfaces. A strong configuration management effort is necessary for newly developed environment types because they must be reused among many programs.

3) Environment types must be well-designed objects.

The environment type creator must be motivated to design the environment types in an object-oriented fashion. An environment type is an autonomous and persistent object providing both the necessary properties about type and operations required to manage tool data of the type. Users of environment

types use these objects in object-oriented development. To support object-oriented development, the environment types must be well-designed objects themselves, providing all the necessary properties and operations with the independency from tool and persistency to keep the state of the objects.

4) Environment types must provide standard interfaces for tool data.

Environment types provide all operations required to manipulate their tool data. These operations are interfaces that serve as standards of methods that are applied to the object. Every tool or application uses this standardized set of operations to manipulate its tool data. As such, the operations must be rich, sufficient, robust, and efficient. The environment type creator is the only person who sets such a standard for the future user of the environment type. By restricting the manipulation of tool data (by allowing given sets of operations only), the interface to tool data can be consistent, that is, the operations defined in the environment type can be the reused among tools in a consistent manner.

5) Environment types must support good abstraction.

Environment types are used to provide better modularity and abstraction for shared data structures and operations. The level of abstraction must support conciseness and generality to support widely used tool data. More specific types can be developed from general environment types as necessary. The environment adaptor plays an important role in adapting generalized environment types (in higher levels of the type lattice) and bringing new, specialized environment types for project-specific purposes. In such cases, the environment type adaptor creates a new set of environment types in the project-specific environment type library. In any level of type creation, the

environment type creator must provide an appropriate level of abstraction for the environment type.

The explicit polymorphism of this typing mechanism helps such environment type development with excellent abstraction. The generic clause of the environment type definition provides a mechanism for defining environment type with type parameters. The project-specific environment type using actual type parameters can be defined later. Such parameterization gives the environment type creator more flexibility to tune to the best level of abstraction.

6.1.1.2 Guidelines for the Environment Type User

The user of the environment types – the software developer for application software and the tool builder – develops software for tools or applications using catalogued environment types in the environment type library. The environment type creator also uses existing environment types to define new types and to write the code part of the operations.

When environment types are used, software development can be improved in several ways as discussed in later sections (sections 6.3 and 6.4 of this chapter); however, the software developer must use the environment types for their software development. When developing software using environment types, the following guidelines are suggested.

1) The environment type must be used as much as possible in a project-wide scope.

Without environment types, the developer must design data structures and relevant procedures. When these data structures are shared with other software, the developer must maintain consistent control over them. To

integrate several tools, it is necessary to maintain the consistent management of tool data among them.

The environment types provides necessary codes for the management of the tool data used in the objective software as the reusable software module. With the encapsulation and abstraction, the manipulation of a tool data is forced to be consistent. The reusability can be greatly enhanced by using environment types.

Both tool builder and application programmer must be highly motivated to use the available environment types. Yet development efforts for common data structures among various tools can be eliminated by using environment types. Accordingly, management efforts are necessary to enforce and reward as much as possible the use of environment types.

2) The user of environment types must learn the standard interface for the environment type (operations).

To use the existing environment types, it is necessary for the user to learn about their functions. To make this task easy, the environment type must be designed well enough to provide the necessary information about the structure and behavior of the environment types. In the beginning, it can be a burden for the user to learn environment type use, but one can be more productive once familiar with it.

3) Object-oriented development must be encouraged.

Since the environment types are objects, the object-oriented development methodology is highly recommended for the productive software development using the types. Tools or application programs use these environment types as objects. The user of the environment types must be encouraged to follow an object-oriented paradigm to design and implement

the software. The typed software engineering environment enhances object-oriented methodology with more automated support. Management efforts must provide necessary motivation to use the object-oriented development.

4) More parameterized programming must be supported for software design. To achieve maximum use of environment types, the design for the tool or application program needs to be highly parameterized. The general design criteria of cohesion and coupling can be applied. More modularization of design must be supported. The object-oriented design can provide such methodology.

6.1.2 Pros and Cons in using Environment Type System

By using environment types in software development process, the developer can enhance the reusability of software tool data, which results in less object size of the software system, and leads to reliable and efficient software systems.

Through the environment type management facilities provided by the software engineering environment, the integration of more tools is feasible and easy, thereby the developer can select the best tools for a specific purpose and phase of the software development. More powerful development activities can be achieved by integrating good tools smoothly. Also, developers can concentrate on their expertise area without concern over developing interfaces of tool data. This results in more productivity.

The entity management facility supports the entity-relation model. This facility can provide systematic services to map real-world data with more manageability. More powerful documentation and source program management can be achieved with an appropriate model.

The environment type management facility is an added layer between the host operating system and user. More computing resources may be required and the possible system slowdown can be expected. However, the benefits from the typing mechanism can easily compensate for this cost during software development.

6.1.3 More practical support for the Object-Oriented Development of Tools

The typing mechanism of the software engineering environment provides dynamic type evolution, management of autonomous objects, message handling through the message queue, concurrency for the tools, and persistency. Such a facility has been crucial in implementing the true object-oriented system. This environment type management system provides the practical support for object-oriented development in the software engineering environment. By using the environment type system, tool data can truly represent functions of the real world objects.

The environment type has autonomous control over tool data objects. Tools send messages rather than sending control to tool data objects. The messages are handled by the environment type management system independent of the tools. Tools are concurrent entities in the environment system. Therefore, all entities of environment type, tool data, message, and tool are independently controlled to provide true autonomous management of objects.

The type inheritance supports full inheritance through specialization, partial inheritance through content inheritance (content field defined as environment type), and multiple inheritance using more than one environment types specialized. Inter-application communication is

supported through tool data. The environment types are not language-specific and are therefore independent of the compiler.

With abundant tools developed using tool data objects for each phase and activity of the software development, a good object-oriented development can be achieved. When equipped with a typing mechanism, the environment system is an underlying system to provide the necessary and sufficient facility for an object-oriented design methodology in a systematic way. It is the user's responsibility to enrich the environment with proper tools for object-oriented development.

6.2 Implementation Analysis: Type System Builder's View

To build a software engineering environment system with a typing mechanism, it is necessary to consider several factors. The analysis in this section examines the necessary requirements to implement a typed software engineering environment and reveals the feasibility of implementing such a system in a software engineering environment. The host language and underlying support system chosen provide a powerful set of interfaces although they have potential compatibility problems. For implementing a type system in various environments with host languages other than Ada, compatibility issues of host language and host environment with the environment type definition language are examined.

6.2.1 Feasibility Issue

The typed software engineering environment used in this research can be implemented in many ways. To provide a variety of facilities discussed in this research such as dynamic type evolution, object management support, entity-relation model, persistency, and concurrency, it is necessary for the host's underlying system to provide and support the powerful resource

management facilities of concurrency, shared memory, and the portable environment. These facilities are basic building blocks for constructing a type system in the software engineering environment.

6.2.1.1 Concurrency and Shared Resources

To support dynamic evolution of the environment types in the software engineering environment, both the definition of the type and instantiation of the type must be performed in the environment during runtime. The environment type creator defines new environment types using the environment type management system while the environment type user instantiates tool data from the existing environment types.

The type system for the software engineering environment includes various environment objects:

1) the subcomponent systems of the environment type management system such as ETDP, tool compiler, environment type library manager, and tool data manager,
2) the environment type monitors for each environment type catalogued, and
3) the tools. Every environment object is the runtime object, which runs as an independent, yet concurrent, entity.

Concurrency support for the software engineering environment is vital in order that the environment objects implemented as concurrent entities can be invoked dynamically. To provide concurrent objects of the environment types, the software engineering environment should provide a rich facility such as CAIS. The interprocess communication facility is also necessary to provide message-handling independently of the tools.

The concurrency of the environment objects must be supported by a process level, not a task level. If these environment objects are Ada tasks, the type system for the software engineering environment can provide only a predefined number of tools and environment types that cannot meet our objectives to define the environment types and tools dynamically.

The environment types and tool data are shared resources in the software engineering environment. To provide an appropriate resource-sharing mechanism among concurrent objects, it is necessary to support the access to common data structures of the node model for environment objects. This can be implemented in many ways, including file systems and shared memory management.

It is our observation that both requirements – concurrency and shared resource management – are available in current programming technology. The typing mechanism for the software engineering environment can facilitate such requirements without much difficulty.

6.2.1.2 Portability of Environment

When the environment types and tool data are distributed over different host environments, it is necessary to have a common environment type management system. Such a management system is host-environment-dependent although the interfaces to the environment type usage must be consistent. A portable environment such as CAIS provides consistent interfaces among different host systems to support common structures on top of CAIS. The type system for the software engineering environment must be built on the portable environment to support consistent interfaces.

Portability is also important to distribute tools or tool data developed in one software engineering environment to another environment. The tool

data used in one environment need to get the support of their environment types in a different environment to maintain consistent behavior. The environment types as environment objects must be transportable. In fact, many of the environment objects such as environment types, tool data, environment type library manager, and tool data manager are required to be portable. The portability of the underlying environment for the environment type system can guarantee the good portability of those objects.

6.2.2 Compatibility Issue

In this research, Ada has been used to implement the environment objects, and CAIS was the underlying system on top of which the type system was built. Ada provides strong typing with a static binding mechanism, packaging, and tasking. CAIS provides well-defined interface sets for a transparent and source-level portable environment.

The type system for the software engineering environment can be implemented using a different host language and host environment. However, it is important to examine the compatibility of the environment type processing languages which are the environment type definition language and the tool data manipulation language, with the host language and environment. There can be potential difficulties in implementing the type system because of rigidness of the host language, lack of an abstraction mechanism in the host language, lack of packaging and specification capability, or lack of concurrency and portability support of the host environment. The required characteristics for the host languages and host environments to support compatibility are examined below.

6.2.2.1 Host Language

To improve software quality and productivity, a programming language evolves to provide more automated support to the software developer. Some of this evolution results in strong typing, static binding, packaging, object-orientation, and specification capability with an abstraction mechanism. The benefits from using Ada such as strong typing, static binding, packaging, and modularity must be preserved where the dynamic environment type management can augment the power of Ada for the software development.

Ada supports strong typing to give the compiler more information to process source code at compile time and to let the programmer avoid potential errors in earlier phases of software development. The strong typing of the host language makes the environment type processing language less compatible because environment types are not the types of the host language and eventually need to be represented in the host language types. The type casting such as coercion and unchecked type conversion of Ada can be a tool for the host language compatible representation for the environment types. It is important that the host language has strong typing with a powerful type-casting mechanism to maintain both productivity support and compatibility.

Static binding of the host language is also an important factor in supporting a productive software development. The dynamic definition and use of the environment types must be supported with dynamic binding of the types in the software engineering environment. The compatibility problem between two different binding strategies must be resolved in an appropriate way. The environment type library plays an important role to manage dynamic entities. At compile time of the environment type definition code, the ETDP uses the environment type library manager to get dynamic services

for the query and catalogue of the environment types. Also, the tool compiler uses the environment type library manager in compile time to generate augmented tool code that will request services for the environment types.

The abstraction mechanism and information hiding through packaging and specification are important for object management. Since the environment types are well-abstracted data, the host language must provide such a capability. Compatibility of the abstraction capability between the environment type definition language and the host language must be resolved by a powerful abstraction mechanism of the host language. Ada provides such a capability well.

6.2.2.2 Host Environment

To implement environment objects as shared and autonomous objects that support the object-oriented paradigm and maintain entity-relation capability, the host environment needs to be compatible with both object-oriented and entity-relation modeling capability. Concurrency support, shared memory support, and interprocess communication support are basic requirements for providing compatibility with environment type processing.

The host environment must support concurrent process management, which provides portable interfaces to manage process handling. Since the environment objects are concurrent entities, the management of those concurrent objects with necessary portability must be compatible to the host environment. The Ada environment, which is equipped with only task-level process management, cannot provide the dynamic process management required in the environment type management system. If the host environment cannot support a portable set of interfaces for process

management, the environment type management system cannot be compatible with the various host environments.

The node model that constructs the environment type structure is an important mechanism that provides dynamic object management with the necessary entity-relation capability. The node model is a shared data structure in the environment type management system. The persistency of such a structure must be supported in the host environment, which provides the set of interfaces to access to the node model in shared memory. Such shared memory management must be supported in lower levels of the software engineering environment layer, such as the virtual operating system or environment support layer of Figure 2.3.

The message-handling that is independent of the language support environment must be implemented through a separate mechanism such as tool data manager. The actual message is sent through an interprocess communication mechanism. The host environment must be compatible to provide such capability.

The host environment must be equipped with a portable set of interfaces to manage process management, node modeling, entity-relation modeling, shared memory, and interprocess communication. Portability is an important aspect of the compatibility issue because software environment objects must be transportable among different host environments in different machines.

6.3 Software Quality Improvement

Environment types are the reusable software components that provide well-defined structures for and behavior of environment tool data objects; in essence, they categorize the environment tool data. Tool data instantiated

from environment types serve as autonomous servers to integrate various tools. Tool data defined in a software program possess the necessary data structures as well as the well-defined code to operate them. Such tool data in a conventional environment are actually part of the software program.

Although the quality of the software is difficult to measure, it is possible to analyze the factors that increase quality. With this in mind, we maintain that the typed software engineering environment improves the quality of the software developed in such an environment. For one thing, the restriction imposed by the typing mechanism in defining and using the environment type results in less variation in the manipulation of tool data. Also, reusability of environment types increases the reliability of the reused software component, which in the typed software engineering environment is tool data. Less diversified and more specialized roles for the software design process enhance the quality. Quality improvement of the software developed in the typed software engineering environment is analyzed in this section with regard to specialization of development activity, reusability, and reliability.

6.3.1 Specialization of Development Activity

Before using of the environment types in software development, the software developer must be well aware of the design of the interfaces to the tool data as well as the informal protocols among related tools that use the tool data. The developers of such tools must share the same information and knowledge to use the tool data correctly. As tools are often not developed in the same group of a project, incomplete information about their use can result in their misuse. The developer must devote his efforts to understanding and writing consistent code so that tool data can be correctly manipulated.

Once the environment types for the tool data are used, the software developers can use the interfaces to manipulate tool data correctly by automated support of the environment types and concentrate their expertise in software development while maintaining correct use of the tool data. The software developer who is expert in a certain area of the development can improve the quality of the software without being distracted by the usual problems with the tool data interfaces.

6.3.2 Reusability

The software developer can use reusable software components in software when those components are available and accessible. To be reusable, the components must be abstracted well and be available. The inhibiting factors of the reusability include the difficulties in representing them, the language dependency, the accessibility when necessary, and the heterogeneous design or implementation methodology. A special mechanism is necessary to support a practical reusability. [Biggerstaff and Richter 1987; Agresti and McGarry 1988; Tracz 1987]. A supporting metrics analysis is presented in Appendix B.

The environment types are reusable and well abstracted as autonomous objects. The tight coupling of the tool data with their interpretation through defined operations makes the tool data reusable in development of related tools. The environment types are in factored form to provide parameterized interfaces that support reusability. The environment types are language independent to provide portable interfaces. Uniform management of the tool data instantiated from environment types supports reusability. Such a dynamic environment type evolution enables gradual

development of the environment types with a controlled degree of abstraction.

The environment types are developed through thorough study of the function and use of the types for various potential users. Once they are catalogued, the environment types provide consistent services to the tool using the tool data. The services provided by the environment types are reliable and of good quality. Reusability of environment types provides the well-understood methods to manage such tool data so that the software developer can concentrate efforts on other parts of the development. These practical supports for reusability enhance the quality of resulting software. The accessibility to the reusable software motivates developers to use them.

6.3.3 Reliability

Because the environment types must be reliable, they are developed and catalogued with extensive testing. When reliable operations of the environment types for the tool data are used, the tool itself can be more reliable in the interfaces to the tool data. The possible errors caused by the mis-handling of the tool data are avoided systematically. Overall reliability of the resulting software is increased when the software developer uses more environment types in the software development. The reliability improvement contributes to the quality of the resulting software.

The environment types catalogued in the software engineering environment provide a set of standardized services for tool-data manipulation. Standardization of the interfaces provides fewer errors in the design phase and enhances the reliability of the software so that the quality of the software is improved.

By using well-understood environment types, the software developer can manage the developing software well. Improvements in manageability of the software resources under development result in more reliable software.

6.4 Improvement in Productivity

The productivity is defined as the ratio of outputs produced to inputs consumed by software development activity [Boehm 1987]. We use inputs as the costs consumed in the software development phases of objective softwares. The cost involving management of environment types in the software engineering environment can be assumed relatively constant for the software project when a rich set of environment types are established.

Using typing mechanisms for tool data in the software engineering environment can improve the productivity of the software primarily by reducing costs during software development. Using environment types, the software developer can use tools more efficiently by integrating them better. The objective software includes well-defined environment types and can manage the tool data with smaller size software. The design process of the software becomes easier because good tools can be selected more easily for that specific purpose. Standardization and consistency of tool data management reduce testing efforts for the tool data interfaces. Improved reusability provides better productivity. Maintenance costs for the developed software can be less because of less variation in the software components involved.

Productivity improvement by using environment types in software development is analyzed below with regard to automation in software development, consistency, and cost. The analysis includes the efforts related to static analysis, dynamic analysis, and management viewpoint for the productivity as well.

6.4.1 Automation

Tools for software development can be integrated more easily by using appropriate environment types available. The software developer can use more tools for his software development activity by such tool integration. This enables more automation of the software engineering environment. Productivity can be increased by using more automated tools with improved integration of tools.

6.4.2 Consistency

The software developer can manage tool data in a consistent way by using operations defined in the environment types. More restrictions are imposed on tool data manipulation by such operations and, therefore, less flexibility is allowed. The software developer uses only a standard set of interfaces to manipulate tool data. This standardization enhances the consistent management of tool data among tools using them, which is important to reduce any potential errors at early stages of the development.

6.4.3 Cost

The software using environment types is not required to define the code to manipulate tool data; in fact, it becomes smaller in size. There are only two pieces of code in the developing software: one to send messages to request services to manage tool data, and a second to define its own specific functions. The usual interface code for the various tool data is eliminated to reduce the size of the software.

Smaller size for the software requires less work for design and coding. Moreover, the testing effort for a smaller size of software can be less costly. Productivity of the software development that uses more well-defined

environment types can be increased if the software engineering environment provides a rich set of environment types for a variety of uses.

6.4.4 Static Analysis Effort

Static analysis efforts in the software development examines the effect of software development regardless of executability. It includes statistical profiling, cost and resource estimation, and design process.

The statistics on the use of environment types can be profiled to analyze and improve the quality of the environment types. By improving the efficiency of the popular environment types and introducing more appropriate environment types, productivity of the software development can be increased. The software engineering environment can be more powerful through this process of environment type evolution.

The cost estimation and resource estimation can be more visible because more reusable environment types for the software are available in the software engineering environment.

The design process becomes more standardized because several reusable environment types can be assembled for routine functions. Also the software programs become smaller in size if more reusable environment types are integrated.

The gains outlined in the above static analysis contribute to the productivity of the software development in the software engineering environment.

6.4.5 Dynamic Analysis Effort

The dynamic analysis effort for the software development examines the software development during and after execution of the software developed. The development activities of the software are more convenient if the

software engineering environment provides a rich set of reusable environment types. Simulation of the software is feasible at an early stage of the design. The tracing or debugging of the embedded errors in the software can be easier because the interfaces to the tool data are more reliable, and the coverage for the testing can be narrower. Tools used for the development of the software can use the tool data more reliably.

6.4.6 Management Viewpoint

The configuration management can be improved by the relation structure of the environment types. Relation description of the environment types enforces the existence of certain objects linked to the relation upon instantiation of tool data. This mechanism helps the configuration management become more automated.

Information management for the software components can be improved with the relation structure also. Necessary documentation can be enforced by defining such environment types with a mandatory relation field to have the documentation type.

When various environment types for the data dictionary, documentation, code object, specification, and test data are supplied in the software engineering environment, the software engineering object can be manageable in more consistent and automated way.

The improvement in productivity estimated in this section can be critical when the software development cycle is short. The initial cost to accumulate well-designed environment types and collect powerful tools can be high, and the execution time of the typed software engineering environment system can be slower. However, the quality of the resulting software and the gain in productivity can easily compensate for the initial

cost. The parameterization of the abstraction mechanism for the environment type is crucial to provide a good, reusable software component. Dynamic environment type evolution is necessary to enable the software engineering environment to be self-evolving and constantly improving.

7. CONCLUSION

A software engineering environment is a well-incorporated system to provide economical software development with a rich set of tools and supporting methodologies. The characteristics of software engineering environments reveal many essential requirements. The architectural view for the software engineering environment system provides a hierarchical structure of layers where the functions and the roles of each subcomponent are placed. Accordingly the roles of software developers are categorized for various activities and phases in the software development life-cycle.

To model the tools and tool data with modularity, good abstraction, and information hiding, the object-oriented development paradigm is summarized, with emphasis placed on autonomous and persistent requirements for the software tool data in software engineering environments.

The typing mechanism for the software engineering tool data utilizing explicit polymorphic types supports good integration of tools in the software engineering environment. A software engineering environment equipped with such a typing mechanism enhances the quality of the software system and the productivity of the software development process. An environment type processing language that defines and manipulates environment types can robustly model real-world software data with the object-oriented paradigm. The entity management system embedded in this typing system provides the entity-relation model that is essential in many software practices.

The environment type management system provides necessary services for several subsystems such as the environment type definition

processor and the tool compiler. This type management system is equipped with the independent subsystem of the environment type library manager and tool data manager that services requests from various sources in the software engineering environment.

The design of such a typing mechanism in a software engineering environment identifies necessary subsystems with their required functions as well as the relationships among them.

The analysis of the typing mechanism from the viewpoint of the use of a typed software engineering environment, the implementation of the system, and quality and productivity improvements in software development, is also presented.

7.1 Contributions

In this research, the typing mechanism for the software engineering tool data is believed to provide the following benefits:

1) efficient management of tool integration in the software engineering environment,
2) maximum reusability of code to manipulate the tool data which provides consistent and standardized services,
3) affordable and manageable representation of software tool data for the various software resources,
4) practical support for object-oriented development of software systems with concurrency, autonomous objects, language-independency, and dynamic type evolution,
5) increased quality and productivity in software development process through reusability, consistency, cost-savings, and better tool manageability.

7.2 Further Research

The prototype system implementing the typing mechanism in the CAIS environment is near completion. The environment type definition processor is working in the CAIS at this point. A robust system for the software engineering environment typing mechanism currently remains for further work. Upon completion of the implementation of the type system, a more objective measurement must be investigated with the criteria given in Chapter 6.

Dynamic type modification for environment types in software engineering environments is needed as type evolution continues. Necessary underlying mechanisms for type modification and generalization is a topic for further research. The transaction mechanism and triggering mechanism are also the future research topics.

REFERENCES

Aho, A., and Ullman, J. 1979. *Principles of Compiler Design,* Addison-Wesley, Readings, Massachusetts.

Agresti, W., and McGarry, F. 1988. The Minnowbrook workshop on software reuse: A summary report. *IEEE Tutorial on Software Reuse: Emerging Technology* by W. Tracz, IEEE.

Archer, J. Jr., and Devlin, M. 1986. Rational's experience using Ada for very large systems. In *Proceedings on First International Conference in Ada programming Language Applications for the NASA Space Station* (NASA), pp.B2.5.1-B.2.5.12.

Barnes, B., Durek,T., Gaffney, J., and Pyster, A. 1987. A framework and economic foundation for software reuse. In *Proceedings of Workshop on Software Reusability and Maintainability* (Oct.).

Basili, V., Barley, J., Joo, B., and Rombach, H. 1987. Software reuse: A framework. Minnowbrook Workshop on Software Reuse.

Basili, V., and Rombach, H. 1988. The TAME project: Towards improvement-oriented software environments. *IEEE Transactions on Software Engineering*, vol. 14, no. 6 (June).

Bassett, P. 1987. Frame-based software engineering. *IEEE Software,* vol. 4, no. 4 (July).

Beacker, R. 1988. Enhancing program readability and comprehensibility with tools for program visualization. In *Proceedings of 10th International Conference on Software Engineering* (Singapore, Apr.).

Biggerstaff, T., and Righter, C. 1987. Reusability framework, assessment, and directions. *IEEE Software* (July), pp.41-49.

Boehm, B. 1981. *Software Engineering Economics*, Prentice-Hall, Inc., Englewood Cliffs, NJ.

Boehm, B. 1987. Improving software productivity. *IEEE Computer*, vol. 20, no. 9 (Sept).

Booch, G. 1983. *Software Engineering with Ada*. Benjamin/Cummings.

Booch, G. 1986. Object-oriented development. *IEEE Transactions on Software Engineering*, vol. SE-12, no. 2 (Feb.).

Buhr. R. J. A. 1984. *System Design with Ada*. Prentice-Hall, Inc., Englewood Cliffs, NJ.

Burton, B., Aragon, R., Bailey, S., Koehler, K., and Mayes, L. 1987. The reusable software library. *IEEE Software*, vol. 4, no. 4 (July).

Buxton, J. 1980. *Requirements for Ada Programming Support Environments, STONEMAN*, U.S. Dep. Defense (Feb.).

Buzzard, G., and Mudge, T. 1985. Project-based computing and the Ada programming language. *IEEE Computer*, vol. 18, no. 3 (Mar.).

CAIS 1986. Military Standard Common Ada Programming Support Environment (APSE) Interface Set (CAIS), DOD-STD-1838, Washington D.C.: U.S. Dep. Defense (Oct.).

Cardelli, L., and Wegner, P. 1985. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, vol. 17, no. 4 (Dec.).

Cheatham, T. Jr., 1984. Reusability through program transformations. *IEEE Transactions on Software Engineering*, vol. SE-10, no. 5 (Sept.).

Chikofsky, E. 1988. Software technology people can really use. *IEEE Software*, vol. 5, no. 2 (Mar.).

Chikofsky, E., and Rubenstein, B. 1988. CASE: Reliability engineering for information systems. *IEEE Software*, vol. 5, no. 2 (Mar.).

Cox, B. 1984. Message/Object Programming: An evolutionary change in programming technology. *IEEE Software*, vol. 1, no. 1 (Jan.).

Cox, B. 1986. *Object Oriented Programming, an Evolutionary Approach*, Addison-Wesley, Reading, Massachusetts.

Dart, S., Ellison, R., Feiler, P., and Habermann, A. 1987. Software Development Environments. *IEEE Computer*, vol. 20, no. 11 (Nov.).

DEC (Digital Equipment Corporation) 1984. *User's Introduction to VAX DEC/CMS*, Digital Equipment Corp., Maynard, Mass.

DoD (U.S. Department of Defense) 1983. Reference manual for the Ada® programming language, ANSI/MIL-STD-1815A. Washington D.C.: Department of Defense (Jan.).

ESPRIT (European Strategic Program for Research and Development in Information Technology) 1986. *PCTE: A basis for a portable common tool environment, in Functional Specifications*. 4th ed., vol. 1, Commission of European Community.

Goguen, J. 1984. Parameterized programming. *IEEE Transactions on Software Engineering*, vol. SE-10, no. 5 (Sept.).

Goguen, J., and Moriconi, M. 1987. Formalization in programming environments. *IEEE Computer*, vol. 20, no. 11 (Nov.).

Goldberg, A. 1980. *Smalltalk-80: The Interactive Programming Environment*. Addison-Wesley.

Goldberg, A., and Robson, D. 1983. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley.

Habermann, A., and Notkin, D. 1986. Gandalf: Software development environments. *IEEE Transactions on Software Engineering*, vol. SE-12, no. 12 (Dec.).

Hailpern, B., and Nguyen, V. 1987. A model for object-based inheritance. In *Research Directions in Object-Oriented Programming*, Eds. Shriver and Wegner, MIT Press, Cambridge, MA.

Halbert, D., and O'Brien, P. 1987. Using types and inheritance in object-oriented programming. *IEEE Software*, vol. 4, no. 5 (Sept.).

Hantler, S., and King, J. 1976. An introduction to proving the correctness of programs. *ACM Computing Surveys* (Sept.), pp. 331-353.

Houghton, R. C. Jr. 1987. Characteristics and functions of software engineering environment: An overview. *ACM SIGSOFT Software Engineering Notes*, vol. 12, no. 1 (Jan.).

Hudson, S., and King, R. 1988. The Cactis project: database support for software environment. *IEEE Transactions on Software Engineering*, vol. 14, no. 6 (June).

IEEE 1983. An American National Standard: IEEE Standard Glossary of Software Engineering Terminology, IEEE, NY

Kaiser, G., and Garlan, D. 1987. Melding software systems from reusable building blocks. *IEEE Software*, vol. 4, no. 4 (July).

Kernighan, B., and Mashey, J. 1981. The UNIX programming environment. *IEEE Computer* (Apr.), pp.12-24.

Kim, W., Banerjee, J., Chou, H., Garza, J., and Woelk, D. 1987. Composite object support in an object-oriented database system. In *Proceedings of ACM Conference on Object-Oriented Programming Systems, Languages and Applications* (Orlando, FL, Oct.).

Kramer, J., Oberndorf, P. Long, J., Robinson, R., Roby, C., Chludzinski, J., and Clouse, J. 1985. The CAIS Reader's Guide. *IDA Memorandum Report M-150*. Institute for Defense Analyses (Dec.).

Lamsweerde, A., Delcourt, B., Delor, E., Shayes, M., and Champagne, R. 1988. Generic lifecycle support in the ALMA environment. *IEEE Transactions on Software Engineering*, vol. 14, no. 6 (June).

Lenz, M., Schmid, H., and Wolf, P. 1987. Software reuse through building block. *IEEE Software*, vol. 4, no. 4 (July).

Levine, D. 1988. Toward the production and application of an environmental type definition processor. Master's Thesis, Arizona State University (May).

Lieberherr, K., and Riel, A. 1988. Demeter: A case study of software growth through parameterized classes. In *Proceedings of 10th International Conference on Software Engineering* (Singapore, Apr.).

Lindquist, T. 1988. *Lecture Notes from CSC 591: Fundamentals of Software Engineering Environments*. Class Notes of Computer Science Course CSC 591, Arizona State University.

Lindquist, T., and Jenkins, J. 1988. Test-case generation with IOGen. *IEEE Software*, vol. 5, no. 1 (Jan.).

Lindquist, T., Lawlis, P., and Levine, D. 1987. Typing information in a software engineering environment. In *Proceedings of Sixth International Conference on Entity Relationship Approach* (New York, NY, Nov. 11-13), pp.165-179.

Martin, C. 1988. Second-generation CASE tools: A challenge to vendors. *IEEE Software*, vol. 5, no. 2 (Mar.).

Meyer, B. 1986. Genericity versus inheritance. In Proceedings of OOPSLA'86, special edition of *SIGPLAN Notices*, vol. 21, no. 11, ACM (Nov.).

Meyer, B. 1987. Reusability: the Case for object-oriented design. *IEEE Software* (Mar.), pp.50-64

Meyers, G. 1979. *The Art of Software Testing*, John Wiley & Sons.

Meyers, R., and Parrish, J. 1988. The Macintosh Programmer's Workshop. *IEEE Software*, vol. 5, no. 3 (May).

Müller, H., and Klashinsky, K. 1988. Rigi - A system for programming-in-the-large. In *Proceedings of 10th International Conference on Software Engineering* (Singapore, Apr.).

Narayanaswamy, K. 1988. Static Analysis-based program evolution support in the Common LISP framework, In *Proceedings of 10th International Conference on Software Engineering* (Singapore, Apr.).

Neck, L., and Perkins, T. 1983. A survey of software engineering practice: tools, methods, and results. *IEEE Transactions on Software Engineering*, vol. SE-9, no. 5 (Sept.).

Notkin, D., and Griswold, W. 1988. extension and software development. In *Proceedings of 10th International Conference on Software Engineering* (Singapore, Apr.).

Oberndorf, P. 1985. *KAPSE Interface Team Public Report:Technical Document 552*. vol. 5, Ada Joint Program Office (AJPO), Naval Ocean Systems Center (Aug.).

Oberndorf, P. 1988. The Common Ada programming Support Environment (APSE) Interface Set (CAIS). *IEEE Transactions on Software Engineering*, vol. 14, no. 6 (June).

Parnas, D. 1972. On the criteria to be used in decomposing systems into modules. *Communications of the ACM* (Dec.).

Penedo, M., and Riddle, W. 1988. Software engineering environment architecture. *IEEE Transactions on Software Engineering*, vol. 14, no. 6 (June).

Penney, D. and Stein, J. 1987. Class modification in the Gemstone object-oriented DBMS. In *Proceedings of ACM Conference on Object-Oriented Programming Systems, Languages and Applications* (Orlando, FL, Oct.).

Perry, D., and Kaiser, G. 1988. Models of software development environments. In *Proceedings of 10th International Conference on Software Engineering* (Singapore, Apr.).

Pressman, R. 1987. *Software Engineering: a Practitioner's Approach*. second edition, McGraw-Hill.

Prieto-Diaz, R., and Freeman, P. 1987. Classifying software for reusability. *IEEE Software*, vol. 4, no. 1 (Jan.).

Ramanathan, J., and Sarkar, S. 1988. Providing customized assistance for software lifecycle approaches. *IEEE Transactions on Software Engineering*, vol. 14, no. 6 (June).

Reiss, S., 1985. Program development systems that support multiple views. *IEEE Transactions on Software Engineering*, vol. SE-11, no. 3 (Mar.).

Rich, C., and Waters, R. 1988. Automatic programming: myths and prospects. *IEEE Computer*, vol. 21, no. 8 (Aug.).

Robson, D. 1981. Object-oriented software systems. *Byte*, vol. 6, no. 8 (Aug.).

Rosenblum, D. 1985. A methodology for the design of Ada transformation tools in a DIANA environment. *IEEE Software*, vol. 2, no. 2 (Mar.).

Seidewitz, E. 1987. Object-oriented programming in Smalltalk and Ada. In Proceedings of OOPSLA'87, special edition of *SIGPLAN Notices*, vol. 22, no. 12, ACM (Dec.).

Steele, G. Jr. 1983. *Common Lisp-The Language*. Digital Press, Burlington, MA.

Swinehart, D., Zellweger, P., and Hagmann, R. 1985. The structure of Cedar, In Proceedings on ACM SIGPlan Symposium of Language Issues in Programming Environments, *SIGPlan Notices* (July).

Taylor, R., Osterweil, L., Wileden, L., and Young, J. 1986. Arcadia: A software development environment research project. In *Proceedings of the IEEE Computer Society Second International Conference on Ada Applications and Environments.*

Taylor, R., and Standish, T. 1985. Steps to an advanced Ada programming environment. *IEEE Transactions on Software Engineering*, vol. SE-11, no. 3 (Mar.).

Teitelbaum, W., Reps, T., and Horwitz, S. 1981. The Cornell Program Synthesizer: A syntax-directed programming environment. *Communications of ACM*, vol. 25, no. 9 (Sept.).

Teitelman, W. 1984. A tour Cedar. *IEEE Software*, vol. 1, no. 2 (Apr.).

Teitelman, W. 1985. A tour through Cedar. *IEEE Transactions on Software Engineering*, vol. SE-11, no. 3 (Mar.).

Tracz, W. 1987a. Ada reusability efforts: A survey of the state of practice. In *Proceedings of the Fifth Annual Joint Conference on Ada Technology and Washington Ada Symposium*, pp.35-44

Tracz, W. 1987b. Reusability comes of age. *IEEE Software*, vol. 4, no. 4 (July).

Tracz. W. 1987c. Software reuse: Motivators and inhibitors. In *Proceedings of COMPCON S'87*, pp.358-363

Tracz, W. 1988. Software reuse myths. *ACM SIGSOFT Software Engineering Notes*, vol.13, no.1 (Jan.), pp.17-21.

Urban, J. and Fisher, D. 1985. Ada environments and tools. *IEEE Software*, vol. 2, no. 2 (Mar.).

Vines, D., and King, T. 1988. Gaia: An object-oriented framework for an Ada environment. In *Proceedings of the 3rd International IEEE Conference on Ada Applications and Environments* (Manchester, NH, May).

Voelcker. J. 1988. Automating software. *IEEE Spectrum*, vol. 25, no. 7 (July).

Walker, J. 1988. Supporting document development with Concordia. *IEEE Computer*, vol. 21, no. 1 (Jan).

Wegner, P. 1987a. Dimensions of object-oriented language design. In *Proceedings of ACM Conference on Object-Oriented Programming Systems, Languages and Applications* (Orlando, FL, Oct.).

Wegner, P. 1987b. The object-oriented classification paradigm. In *Research Directions in Object-Oriented Programming*, Eds. Shriver and Wegner, MIT Press, Cambridge, MA.

Wolf, A., Clarke, L., and Wileden, J. 1985. Ada-based support for programming-in-the-large. *IEEE Software*, vol. 2, no. 2 (Mar.).

Woodfield, S., Embley, D., and Scott, D. 1987. Can programmers reuse software?. *IEEE Software*, vol. 4, no. 4 (July).

Yankelovich, N., Haan, B., Meyrowitz, N., and Drucker, S. 1988. Intermedia: The concept and the construction of seamless information environment, *IEEE Computer*, vol. 21, no. 1 (Jan.).

Young, M., Taylor, R., and Troup, D. 1988. Software environment architectures and user Interface facilities. *IEEE Transactions on Software Engineering*, vol. 14, no. 6 (June).

Young, M., Taylor, R., Troup, D., and Kelly, C. 1988. Design principles behind Chiron: A UIMS for Software Environments. In *Proceedings of 10th International Conference on Software Engineering* (Singapore, Apr.).

APPENDIX A

Environment Type Definitions for the Software Testing Environment

```
-- File <Symbol_table_Env.ENV> contains declaration for SYMBOL_TABLE

with SKELETON_AND_CODE_CELLS,CAIS_LIST_DEFINITIONS;
include STRUCTURAL_NODES;
nests
   environment type PROGRAM_SKELETON specializes STRUCTURAL_NODES
      with
         contents
            DECLARATION_LIST : CAIS_LIST_TYPE;
         relations
            CONTAINS to PROGRAM_SKELETON,
            INSTRUCTIONS to CODE_CELLS cardinality = 1;
   end PROGRAM_SKELETON;

   environment type CODE_CELL specializes STRUCTURAL_NODES
      with
         contents
            CODE : CAIS_LIST_TYPE;
         relations
            TEXT_ORDER to CODE_CELL cardinality = 0..1;
   end CODE_CELL;
end nests;

environment type SYMBOL_TABLE specializes STRUCTURAL_NODES
   with
      contents
         PROGRAM_TREE : PROGRAM_SKELETON,
         INTERMEDIATE_CODE : CODE_CELL;
      relations
         COMPILED_FROM to ADA_SOURCE predefined cardinality=1,
         EXECUTABLE_IMAGE to ADA_IMAGE predefined cardinality=1;
      operation
         procedure ADD_SYMBOL (SYMBOL: IDENTIFIER);
         procedure ADD_PROG_UNIT_CONTOUR (SYMBOL : IDENTIFIER);
         procedure ADD_CODE_CELL (CODE : CAIS_LIST_TYPE);
         procedure FIND_SYMBOL (SYMBOL : IDENTIFIER);
end SYMBOL_TABLE;
```

```
-- File <Program_Env.ENV> contains the declaration for PROGRAM_ENV

with CAIS_LIST_DEFINITIONS;
include SYMBOL_TABLE, SOURCE_PROGRAM;
nests
   environment type SYNTAX_TREE_ENV
      with
         contents
            PARSE_TREE : CAIS_LIST_TYPE;
         attributes
            SIZE : INTEGER;
         relations
            SYMBOL_TABLE_OF to SYMBOL_TABLE
               cardinality = 1;
         operation
            procedure INSERT_SYMBOL (PARSED_LIST : CAIS_LIST_TYPE);
            function  GET_SYMBOL return CAIS_LIST_TYPE;
   end SYNTAX_TREE_ENV;
```

```
end nests;

environment type PROGRAM_ENV
   with
      contents
         SYMBOL_TAB      : SYMBOL_TABLE,
         SYNTAX_TREE     : SYNTAX_TREE_ENV;
      attributes
         CREATOR         : CREATOR_NAME,
         CREATION_DATE   : TIME,      -- set by OPEN_PROGRAM_SKELETON
         UPDATE_DATE     : TIME,      -- updated by any subsequent
                                      -- change
         SIZE            : INTEGER;   -- set by last change
      relations
         COMPILED_FROM to SOURCE_PROGRAM
            with attribute
                     COMPILER       : COMPILER_TYPE,
                     COMPILATION_DATE : TIME
            cardinality=0..1 primary      -- allow 0 cardinality for
                                          -- TEST_PROGRAM
      operation
         procedure SET_PROGRAM_SKELETON (CREATOR : CREATOR_NAME;
                                 SOURCE_PROG : SOURCE_PROGRAM
                                 COMPILER : COMPILER_TYPE;
                                 COMPILATION_DATE : TIME);
         for ADD_SYMBOL use
            procedure INSERT_TOKEN (TOKEN : TOKEN_TYPE;
                                 ID_KIND : ID_TYPE);
         procedure DELETE_TOKEN (TOKEN : TOKEN_TYPE;
                                 ID_KIND : ID_TYPE);
         function IS_TOKEN (TOKEN : TOKEN_TYPE) return BOOLEAN;
         procedure GET_NEXT_TOKEN (TOKEN : out TOKEN_TYPE;
                                 ID_KIND : out ID_TYPE);
         function COMPILER return SOURCE_PROGRAM;
         function COMPILATION_DATE return TIME;
         function CREATOR return CREATOR_NAME;
         function CREATION_DATE return TIME;
         function UPDATE_DATE return TIME;
end PROGRAM_ENV;
```

```
-- FILE <SOURCE_ENV.ENV> contains the declaration for SOURCE_ENV

includes PROGRAM_ENV, SPECIFICATION;
environment type SOURCE_ENV specializes PROGRAM_ENV
   with
      relations
         SPEC_OF to SPECIFICATION
            with attribute SPEC_KIND : SPEC_TYPE
            cardinality = 1,
         TEST_OBJ_OF to TEST_OBJECTIVE cardinality = 1..MAX_TEST_OBJ,
         TEST_PROGRAM_OF to TEST_PROGRAM cardinality =
                                          1..MAX_TEST_PROG;
      operation
         procedure LINK_TEST_OBJ (TEST_OBJ : TEST_OBJECTIVE);
               -- allows incremental definition of env type
               -- TEST_OBJECTIVE
```

```
        procedure LINK_TEST_PROG (TEST_PROG: TEST_PROGRAM);
end SOURCE_ENV;
```

```
with CAIS_LIST_DEFINITIONS;
includes SOURCE_ENV, TEST_OBJECTIVE;
environment type SPECIFICATION
   with
      contents
         SPEC_CONTENT : CAIS_LIST_TYPE;
      attributes
         NUMBER_OF_ITEM : INTEGER,
         CREATOR        : CREATOR_NAME,
         CREATION_DATE  : TIME,
         UPDATE_DATE    : TIME,
         SIZE           : INTEGER;
      relations
         PROGRAM_OF to SOURCE_ENV cardinality=1..MAX_PROG,
         TEST_OBJ_OF to TEST_OBJECTIVE cardinality = 1..MAX_TEST_OBJ;
      operation
         procedure SET_SPEC (CREATOR : CREATOR_NAME);
         procedure WRITE_SPEC_ITEM (SPEC_ITEM : SPEC_ITEM_TYPE);
         function  NEXT_SPEC_ITEM return SPEC_ITEM_TYPE;
         procedure LINK_SOURCE (SOURCE : SOURCE_ENV);
         procedure LINK_TEST_OBJ (TEST_OBJ : TEST_OBJECTIVE);
         procedure DELETE_SPEC;
end SPECIFICATION;
```

```
with CAIS_LIST_DEFINITIONS;
includes SOURCE_ENV, SPECIFICATION, TEST_PROGRAM, TEST_DATA_ENV;
environment type TEST_OBJECTIVE
   with
      contents
         TEST_OBJ_TUPLE : CAIS_LIST_TYPE;
      attributes
         CREATION_DATE  : TIME,
         SIZE           : INTEGER;
      relations
         SOURCE_OF to SOURCE_ENV cardinality = 1 primary,
         SPEC_OF to SPECIFICATION cardinality = 1 primary,
         TEST_PROGRAM_OF to TEST_PROGRAM cardinality =
                                          1..MAX_TEST_PROG,
      operation
         procedure SET_TEST_OBJECTIVE_TEMPLATE
                        (SOURCE : SOURCE_ENV;
                         SPEC   : SPECIFICATION);
         procedure INSERT_TEST_OBJ (TEST_OBJ_CASE : CAIS_LIST_TYPE);
         procedure DELETE_TEST_OBJ (TEST_OBJ_CASE : CAIS_LIST_TYPE);
         function  GET_TEST_OBJ return CAIS_LIST_TYPE;
         procedure LINK_TEST_PROG (TEST_PROG : TEST_PROGRAM);
end TEST_OBJECTIVE;
```

```
includes PROGRAM_ENV, SOURCE_ENV, SPECIFICATION, TEST_OBJECTIVE,
TEST_DATA;
environment type TEST_PROGRAM_ENV specialize PROGRAM_ENV
   with
      relations
         PROGRAM_OF to SOURCE_ENV cardinality=1 primary,
         TEST_OBJ_OF to TEST_OBJECTIVE cardinality=1,
         TEST_DATA_OF to TEST_DATA cardinality=1..MAX_TEST_DATA;
      operation
         procedure SET_TEST_PROGRAM (SOURCE : SOURCE_ENV;
                                     SPEC : SPECIFICATION;
                                     TEST_OBJ : TEST_OBJECTIVE);
         procedure LINK_TEST_DATA (TEST_DATA : TEST_DATA_ENV);
end TEST_PROGRAM;
```

```
with CAIS_LIST_DEFINITIONS;
includes SOURCE_ENV, SPECIFICATION, TEST_OBJECTIVE, TEST_PROGRAM;
environment type TEST_DATA_ENV
   with
      contents
         TEST_DATA_LIST : CAIS_LIST_TYPE;
      attributes
         CREATOR        : CREATOR_NAME,
         CREATION_DATE  : TIME,
         SIZE           : INTEGER;
      relations
         PROGRAM_OF to SOURCE_ENV cardinality=1 primary,
         SPEC_OF to SPECIFICATION cardinality=1 primary,
         TEST_OBJ_OF to TEST_OBJECTIVE cardinality=1,
         TEST_PROGRAM_OF to TEST_PROGRAM
                         cardinality=1..MAX_TEST_PROGRAM;
      operation
         procedure SET_TEST_DATA  (SPEC : SPECIFICATION;
                                   SOURCE : SOURCE_ENV;
                                   TEST_OBJ : TEST_OBJECTIVE;
                                   TEST_PROG : TEST_PROGRAM);
         function  GET_NEXT_TEST_DATA_CASE return CAIS_LIST_TYPE;
         procedure INSERT_TEST_DATA_CASE
                   (TEST_DATA_LIST : CAIS_LIST_TYPE);
         procedure DELETE_TEST_DATA_CASE
                   (TEST_DATA_LIST : CAIS_LIST_TYPE);
         procedure CLOSE_TEST_DATA;
end TEST_DATA_ENV;
```

APPENDIX B

Metric Formulation for the Reusability

Reusable software components can contribute to enhancing productivity and quality, while reducing the cost of software development. To enhance reusability, the developer is urged to: use parameterization for the existing programming, set up the library to maintain the reusable components, and use object-oriented design methodology.

The developer needs to use correct parameters required by the reusable components. These interfaces were not yet created thought in conventional development, where the developer creates necessary subfunctions according to the requirements of the program design. However, to use reusable components, the developer must match the correct parameters. Further, the developer must create new software modules that are potentially reusable by others, which must be well defined and highly parameterized. To achieve this, the library for a specific problem domain must be set up and maintained properly.

The new software development for reusability will employ the top-down approach for high-level design, while the detailed design ideas can be formulated by composing reusable software in a bottom-up fashion. The object-oriented design methodologies must be introduced to the developer at some point to greatly optimize the development activity.

In the new development paradigm that employs reusability, the design process will include:

1) search activities for possibly reusable components for specific purposes,
2) the creation of new reusable components, or
3) the design of functions in conventional way.

Correlations of the Factors in Reusability

We wish to isolate various factors involving reusability in software development and find correlations among them. Once a simple metric is formulated, it will be easy to find the contributing factors for more productive software development. To design a simple model, we assume that we can estimate or have the actual cost data for software development cost both using reusable components and by creating software from scratch. The discussion presented presented below is based on the economics model by Barnes and others [Barnes et al. 1987].

We assume that the total cost of developing target software using reusable components is C^r, and the total cost of developing target software by creating from the scratch is C^s. The reusable software components in the library are $(R_1, R_2,..., R_r)$, where actually used reusable components for target software are $(R_1, R_2,..., R_m)$, $0<m\leq r$. The target software is composed of $(P_1, P_2,..., P_q, R_1, R_2,..., R_m)$. The proportion of $(R_1, R_2,..., R_m)$ in target software by any measurement criteria – probably by LOC (lines of code) – is R%. If we say the cost of developing $(P_1, P_2,..., P_q)$ is Cn, then $Cn = (1-R)\bullet C^s$ since the unit cost of developing a new part of the target software would be the same as that from building from scratch. To use the reusable components, we need to integrate them where the integration cost of $(R_1, R_2,..., R_r)$ into the target software is $B = b\bullet C^s$, and b is the relative cost for integration to C^s. The cost ratio of developing $(R_1, R_2,..., R_r)$ for reuse purposes versus non-reuse purposes is e, and the aggregate average number of uses of $(R_1, R_2,..., R_r)$ is n. The amortized cost for developing reused components originally in the target software is therefore $\frac{(R\bullet C^s\bullet e)}{n}$.

The cost of developing the target software consists of the cost of new parts, the cost of reused parts, and the cost of integration.

$$C^r = Cn + \text{integration cost} + \text{cost of reused components}$$

$$= (1-R)\bullet C^s + b\bullet C^s\bullet R + \frac{(R\bullet C^s\bullet e)}{n}$$

$$= C^s\bullet(1-R+b\bullet R + \frac{R\bullet e}{n}) = C^s\bullet(1 + R\bullet(b-1+\frac{e}{n}))$$

The relative cost of development by reuse to that of creation from scratch:

$$RC = \frac{C^r}{C^s} = 1 + R\bullet(b-1+\frac{e}{n})$$

The productivity of development by reuse to that of creation from scratch:

$$RP = \frac{C^s}{C^r} = \frac{1}{1+R\bullet(b-1+\frac{e}{n})}$$

The break-even number of uses for library : $N_0 = n$

when $1 + R\bullet(b-1+\frac{e}{n})=1$

which is $b = 1 + \frac{e}{N_0}$ or $N_0 = \frac{e}{1-b}$.

Our interests remain in the productivity gain measured as the relative cost of the reuse paradigm to the non-reuse paradigm. Among the many arguments in favor of reusability, the relative cost for integrating reusable software module b and the cost of developing reusable components must be low enough to pay off initial costs such as library set-up cost.

To get positive productivity (RP>1, which is at least as good as conventional development), $b-1+\frac{e}{n} < 0$ or $b<1-\frac{e}{n}$ requires a sufficiently low integration cost if there is at least one reused software component (R>0). Further observation reveals that it is necessary to reach a certain number of break-even point (N_0) when using reusable components to be cost-effective.

To reduce the *integrating cost* and induce the developer to increase reusability for the development activities, we need a good mechanism that can support the creation and management of reusable components in a *highly parameterized* and *automated* manner.