The Alpha Operating System: Programming Language Support

Samuel E. Shipman

TR #88042

Archons Project Technical Report

October 1, 1988

The Alpha Operating System: Programming Language Support

Samuel E. Shipman

October 1, 1988

Archons Project Technical Report #88042

This research was supported in part by the USAF Rome Air Development Center under contract number F30602-85-C-0274. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of RADC or the U.S. Government.

Table of Contents

List of Figures

Abstract

This report describes the programming language support provided for Release 1 of the Alpha operating system [Northcutt 87]. Alpha is an adaptable decentralized operating system for real-time applications, being developed as a part of the Archons project's on-going research into real-time distributed systems.

Alpha's programming language support consists of a set of extensions to the C programming language that provide a high-level interface to the mechanisms provided by the Alpha kernel (such as objects, operation invocation, threads, and capabilities), and allow users to easily define initial configurations of embedded Alpha applications. A rationale for the design of these extensions is given, based on their system context and requirements. Their implementation, which is based on a preprocessor and was completed in a few programmer-months, is also described. Due to the techniques used in their implementation, the tools that constitute the preprocessor implementation provide substantial functionality, yet cost relatively little to produce. The Alpha Release 1 user programming environment was used successfully by personnel outside the project to produce a demonstration application for Alpha.

1 Introduction

This is a report on the programming language support provided for Release 1 of the Alpha operating system [Northcutt 87]. Alpha is an adaptable decentralized operating system for real-time applications, being developed as part of the Archons project's on-going research into real-time distributed systems. Alpha is a new kind of operating system, which is unique in two highly significant ways—first, it is *decentralized*, providing reliable resource management transparently across physically dispersed nodes, so that distributed applications programming can be done as though it were centralized; and second, it provides comprehensive high technology support for real-time applications, particularly supervisory control systems (e.g., industrial automation) which are characterized by predominately aperiodic activities, having critical time constraints (such as deadline) associated with them. Alpha is extremely adaptable so as to be easily optimized for a wide range of problem-specific functionality, performance, and cost.

In the early days of Alpha development, all Alpha application software was written by the researchers building the system and consisted mainly of programs that exercised kernel functionality in order to test it as it was developed. More recently, companies involved in real-time system development have become interested in the system, and one company in particular has developed a substantial application for Alpha. Also, Archons project personnel not involved directly in kernel development have built applications. Because of this, the Alpha cross-development environment was enhanced to support the development of fairly ambitious applications by users who know nothing about the internals of Alpha. The programming language support described in this report is a major component of those enhancements.

Alpha's programming language support consists of a set of extensions to the C language that provide an appropriately high-level interface to the mechanisms provided by the Alpha kernel (such as objects, operation invocation, threads, and capabilities) and allow users to easily define initial configurations of embedded Alpha applications. Issues of programming in the large, as well as programming in the small, are addressed.

The kernel interface is provided by language constructs because the fundamental mechanisms provided by the Alpha kernel directly as part of its programming model are at a sufficiently high semantic level that they correspond directly to object-oriented language abstractions. An objective of the Alpha research is to explore the suitability of using the programming model provided by Alpha directly for building significant real-time systems. Therefore, the language support for that programming model should reflect it faithfully, without adding significant functionality. Furthermore, language-independence was not an important issue for this research prototype system.

The implementation of these extensions is preprocessor-based and was completed in a few programmer-months. Such an implementation was acceptable because the major emphasis of the Alpha research is on operating systems, not programming languages *per se*. However, the tools that comprise the implementation provide substantial functionality considering the relatively small amount of effort involved in producing them, due to the implementation techniques used. The Alpha Release 1 user programming environment was used successfully by General Dynamics personnel to produce a demonstration application for Alpha.

Our own experience, and that of General Dynamics personnel, in developing applications using this system indicates that it represents a successful approach to supporting embedded system development. The General Dynamics programmers are united in asserting that it is easier to develop applications of this kind under Alpha (because of the Alpha programming model and development tools) than under UNIX.

In subsequent versions of the system, currently under development at Concurrent Computer Corporation [Jensen 88], the kernel interface will be more traditional and language-independent due to the demands of compatibility and versatility. Nevertheless, it is apparent that the language-extension approach to providing an interface to high-level kernel facilities holds promise in the world of embedded systems, in which compatibility is less often an issue, and software maintainability is especially critical. The system described herein illustrates the viability of these techniques.

The remainder of this report describes the design and implementation of the programming language support for Alpha Release 1. Chapter 2 outlines the system context (Alpha and the Alpha Distributed System Testbed) and the requirements for the language support. Chapter 3 is a detailed description of the language, Chapter 4 gives a rationale for the language design, and Chapter 5 describes the implementation.

2 Context and Requirements

This chapter describes the system context in which the need for the Alpha programming language support arose (Section 2.1), and the requirements for the language support derived from that context (Section 2.2).

2.1 System Context

This section explains the system context of the Alpha programming language support effort.. It gives background information necessary to understand the language requirements, design, and rationale.

Section 2.1.1 outlines the programming model provided by Alpha. Section 2.1.2 briefly characterizes the system execution environment (the Alpha distributed testbed system, described in detail in [Northcutt 88c]). It also briefly describes the cross-development environment used to develop Alpha and its applications.

2.1.1 Alpha Programming Model

This is a brief, selective, and rather abstract overview of the Alpha programming model, intended to provide some motivation for the language requirements (Section 2.2) and some background for the language design description (Chapter 3). The details of the language should offer additional insight into the programming model. Other reports provide comprehensive explanations of the Alpha programming model [Northcutt 88b] and its rationale [Northcutt 88a].

Section 2.1.1.1 describes the basic abstractions of Alpha, objects and threads. Section 2.1.1.2 concentrates on thread attributes and exception handling.

2.1.1.1 Object and Threads

The Alpha kernel provides the user with a programming model based on *objects* and *threads*. An object encapsulates data, and the code that manipulates that data, within an individual protection domain. Each object has its own hardware-protected address space. The user designates certain procedures in the object's code as *operations*, each of which has a kernel-protected entry point. Access to these entry points is controlled by *capabilities*, which are kernel-protected names for objects that also encode access privileges for operations on the objects. Each object has a set of capabilities that define its access to other objects. It is straightforward to implement abstract data types as Alpha objects.

Alpha objects are completely passive; one or more threads must be active in an object for computation to occur within the object. A thread embodies a single point of control with its execution context and local state (registers and stack) and special attributes (such as time constraints, atomic transactions, etc.). Threads execute operations on objects, moving from one object to another by executing an *invocation* of an operation on the other object. An invocation is essentially a procedure call across protection domains and sometimes across machine boundaries (remote procedure call). Parameters are passed by value-result. The object on which an operation is being invoked is named in the invoke operation by a capability which the invoking object possesses.

An Alpha object with a single operation and a single thread active within that operation corresponds roughly to a process in a traditional process-based system (such as UNIX). In fact, the Alpha programming model can be thought of as cleaving the traditional process abstraction into two parts corresponding to data (object) and control (thread). Unlike systems such as Mach [Accetta 86], in which multiple "threads" are captive within a single "task," Alpha provides threads that can move among objects by means of operation invocations; in this way, a single activity in the system is represented by a single thread. In Alpha, operation invocation rather than message passing is the means of interaction among objects.

2.1.1.2 Thread Attributes and Exceptions

An Alpha thread has certain attributes that determine how it behaves under various conditions. Examples of thread attributes include:

- the thread's scheduling constraints (part of the information used by the scheduler to resolve contention for CPU resources)
- whether user-specified exception handling code should be executed when a thread encounters an exception, and which code to use (thread exceptions are explained below)
- whether the thread is executing an atomic transaction

A particular thread attribute (an instance of one of the attribute types listed above) applies over a continuous sub-interval of the thread's execution, and all of the thread's attributes are properly nested. In other words, if we imagine a linearized history of the thread's execution represented as a piece of string, and represent attributes as colors applied to this string, then any segment of the string may have more than one color (attribute) associated with it. However, if there are attributes *a* and *b*, such that the thread had attribute *a* before attribute *b*, then the thread cannot have attribute *b* after it ceases to have attribute *a*. This is just the usual concept of nesting, illustrated in Figure 1.

Alpha is a real-time operating system and therefore must take the timing requirements of its application into account when resolving contention for resources; this is called time-driven resource management. In particular, one of the schedulers implemented in Release 1 of Alpha is an instance of a class of time-driven schedulers known as "best-effort" schedulers [Locke 86], which take both the urgency and the relative importance of activities into account in making scheduling decisions. This information is represented as a time-value function [Jensen 75], which specifies the application-determined value to the system of completing a computation as a function of its completion time. The function itself is constrained to be of a particular class specified by a set of parameters that are intrinsically meaningful and capable of expressing a number of desirable modes of behavior of time-critical computations. Section 3.1.2.1 describes time-value functions in more detail.

Atomic transactions are not fully implemented in Release 1 of Alpha, and so are not discussed here.

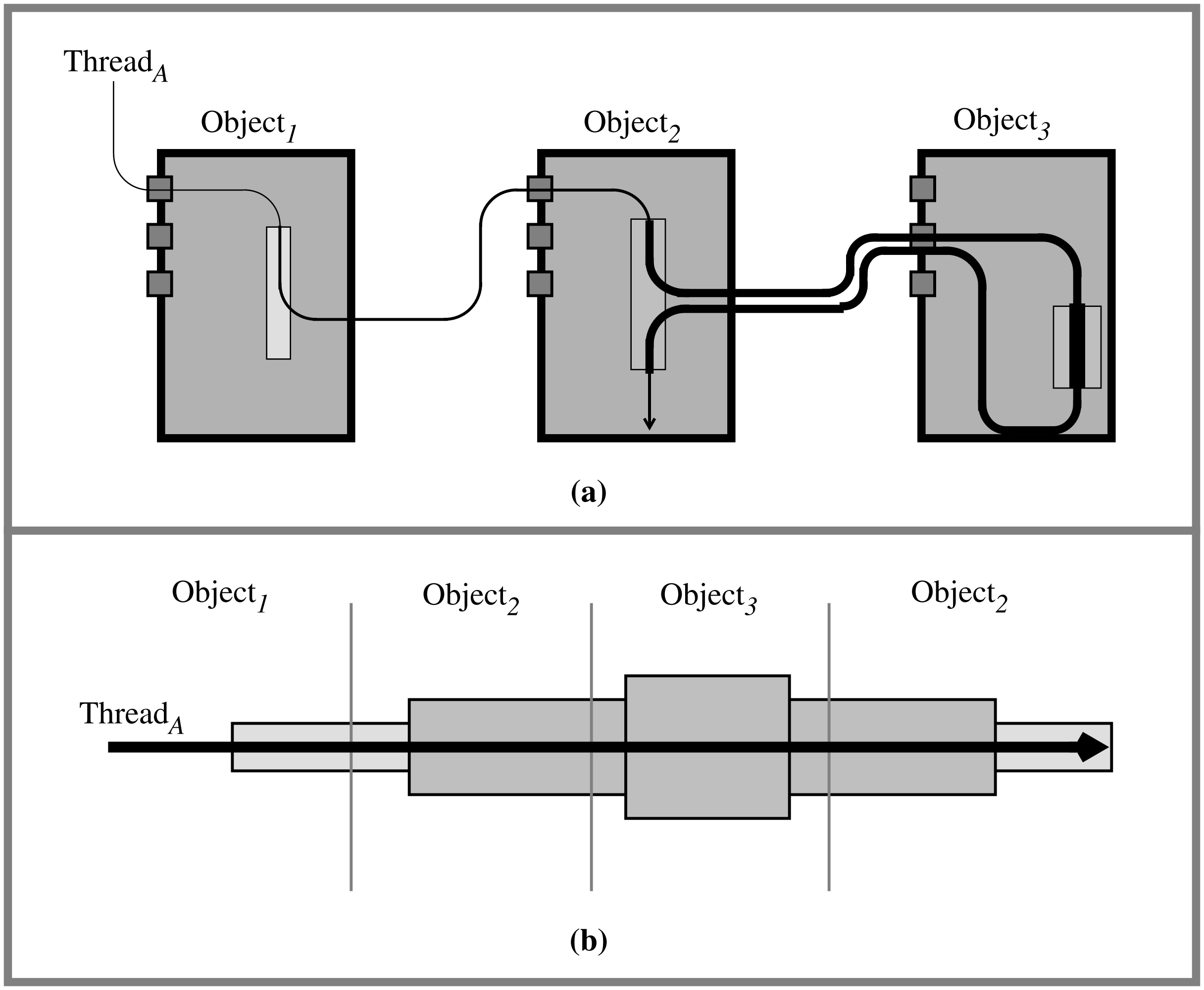

Figure 1: Example of Thread Attribute Nesting

The Alpha Release 1 kernel provides mechanisms for handling exception conditions that must result in the asynchronous diversion of control from a thread's normal execution. A thread exception can occur for one of several possible reasons:

- a thread fails to meet one of its time constraints
- a thread break occurred (e.g., a node through which a thread passed became inoperative, and all sections of the thread beyond the point where it entered that node are automatically aborted by the system)
- a machine error (e.g., dividing by zero)
- others (e.g., atomic transaction abort) to be defined in future versions of the operating system

Alpha allows the application programmer to specify, for dynamically nested intervals of thread execution, routines to handle such exceptions. The kernel coordinates the processing of exceptions and maintains the consistency of kernel resource (such as synchronization objects) in the face of asynchronous transfers of control.

More detail on exception processing from the application's point of view is provided in Section 3.1.2.3. More information on the design, implementation, and rationale for thread exception processing may be found in [Northcutt 88b].

Further details of the Alpha Release 1 kernel interface are discussed in connection with the language design (Chapter 3), when necessary to explain the semantics of the language constructs. A more detailed exposition of the kernel interface is given in [Northcutt 88e].

2.1.2 Execution and Development Environments

Alpha is intended to support supervisory real-time control applications, which are commonly characterized by a single mission—i.e., the entire system cooperates to achieve common goals, rather than having multiple applications competing for resources to achieve divergent goals. Alpha runs on a dedicated collection of interconnected computing nodes that are unified by the instances of the Alpha kernel on each node to act as a single system supporting the application. These characteristics have much in common with embedded systems, and Alpha is designed to support embedded systems.

A high-level view of Alpha's operation is as follows. Alpha currently runs on a distributed system testbed [Northcutt 88c] consisting of multiple computing nodes (modified Sun-2 systems) interconnected by a network (Ethernet). For each node, a load module is built that contains the kernel and the client object types needed for the application. These load modules are downloaded to the testbed nodes from development systems via the network. The system is started and each node boots its load module. On each node, the operating system performs its initialization, then transfers control to the application. The system runs until its mission is completed; individual nodes may fail or be added to the system, but there is continuity of the system and application functions.

Alpha was developed using the software development tools provided by SunOS (Sun Microsystems' operating system derived from Berkeley UNIX) on Sun-3 workstations, along with custom Alpha-specific and general-purpose tools developed by Alpha personnel and running under UNIX and Alpha. (These tools are described in [Northcutt 88d] and in this document). This is a cross-development environment, in which Alpha system and application code is developed on Sun-3 systems running SunOS and downloaded to the Alpha distributed system testbed. Alpha was written mainly in C, plus a relatively small amount of M68000 assembly language.

2.2 Language Requirements

This section defines the requirements for the Alpha Release 1 programming language support. Requirements are based on the system context discussed in Section 2.1, and fall into two major categories: kernel interface support requirements (Section 2.2.1) and configuration support requirements (Section 2.2.2). In addition, there are the following implementation requirements.

The major focus of the Alpha research is on operating systems, not programming languages. Therefore most of the project's resources must be applied to operating systems work. This leads to the requirement that the programming language support not require a high level of effort (in programmer-months) to implement.

However, Alpha must have an application programmer interface and development environment that is readily usable by programmers other than the Alpha kernel developers to build non-trivial application systems. Also, it must be relatively easy to learn and use.

2.2.1 Kernel Interface Support

The programming language support for the Alpha kernel interface must provide the application programmer with a straightforward interface to the kernel, expressed as language constructs at a semantic level appropriate for the intended usage of the Alpha kernel mechanisms. It should specifically address those fundamental mechanisms provided by the Alpha kernel directly as part of the programming model. Other parts of the kernel interface, the "system services," are provided as "system objects," accessible via object invocation (a fundamental mechanism).

One objective of the Alpha research is to explore the suitability of using the programming model provided by Alpha directly for building large real-time systems. Therefore, the language support for that programming model should reflect it faithfully without adding significant functionality.

The language must allow users to define and name object types. (Instantiation is provided as an operation on the ObjectManager system object.) It must allow users to define and name the operations provided by an object type. It must allow symbolic naming of operations, and provide for the automatic, user-transparent mapping of the symbolic names into operation indices used by the kernel. It must allow the user to define operation parameters of arbitrary type (subject only to restrictions imposed by the kernel), and to specify whether a parameter is an input to the operation or an output from the operation

The language must provide a means by which a thread executing within an object can invoke an operation on another object by specifying a capability for the object, the symbolic name of the operation, and the actual input parameters to the operation. It must provide a means by which the output parameters from the operation can be accessed after the invocation is completed. The language must support the association of an object type with a capability for purposes of parameter type checking. It must support the passing of capabilities as parameters in an invocation. It must also provide a means by which an operation can return control to the invoking operation, specify whether the operation is considered to have succeeded or failed, and return an optional status code.

The language must provide a way for the user to specify a time-constraint (i.e., a time-value function) that applies to a continuous interval of execution. There may be multiple such intervals with associated time constraints, possibly nested.

The language must provide a way for the user to specify an exception handler (i.e., a section of code that performs exception processing) that applies to a continuous interval of execution. There may be multiple such intervals with associated exception handlers, possibly nested. The language must define the language-level semantics of thread exceptions and exception processing, based on the mechanisms provided by the kernel.

The language must support programming in the large, in the sense that it should allow separate compilation of object definitions and provide for inter-object type checking of invocation parameters. This facilitates the development of large application software systems.

2.2.2 Configuration Support

As explained in Section 2.1.2, Release 1 of Alpha is intended to support embedded systems particularly well. Because embedded systems typically have a single purpose, or mission, a considerable amount can usually be known *a priori* about the structure of the system.

The initialization of an embedded system, especially a distributed one, can be a tedious and sometime complex process to code. Knowledge of the system's structure can be used to specify an initial configuration of object types, object instances, threads, and access permissions for the system. Automating the initialization of such a configuration relieves the application programmer of a considerable burden. If the application is inherently very static, all the necessary initialization could be described by such a configuration. The user should have the ability to choose any point in a spectrum from completely automated initialization based on a static configuration specification to completely customized initialization directed dynamically by user-written application initialization code.

The programming language support for configuration of Alpha Release 1 applications must provide for the specification of configurations at system-build time such that the initialization of these configurations is automated by the system.

The language must support the definition of system configurations. A configuration consists of node-specific configuration information for each node in the distributed system, and configuration-wide information for the configuration as a whole.

The language must support the specification, for each node, of object instances to be created upon initialization. It must support the specification of initial threads to be created, including, for each thread, the object instance and operation in which it is to be created, with the actual parameters to that initial operation.

The language must support the specification of which object types are to be included in the configuration.

The object instances in the configuration require specific capabilities in order to invoke operations on other application objects, access system services, and create instances of object types. The language must provide for the initial distribution of the capabilities necessary to provide the access permissions specified for objects in the configuration.

3 Language Design

The Alpha Object-oriented Language (AOL) consists of the standard C language plus some extensions that address the requirements stated in the previous chapter. The AOL extensions were designed so that a compiler that accepts source files containing AOL constructs could also accept files containing standard C code. Because the extensions are the focus of interest in this report, they alone are often referred to as "AOL."

The AOL extensions provide the operating system kernel interface to Alpha and allow the user to specify initial configurations of distributed applications under Alpha. AOL has two corresponding top-level constructs: the *object definition*, which embodies the kernel interface constructs and the *configuration definition*, which embodies the configuration constructs. These are block-structured constructs, the bodies of which are delimited by curly braces: {}.

The notation used to specify AOL syntax is defined in Appendix I. Note that, in the syntax specifications, we make no effort to unify AOL syntax with that of the C language—we are only concerned with the AOL extensions in the syntax descriptions. The relationships between AOL constructs and C are made clear in the text of this report.

The kernel interface constructs are discussed in Section 3.1 and the configuration constructs are discussed in Section 3.2.

3.1 Kernel Interface Constructs

The object definition is the top-level AOL construct that embodies kernel interface constructs. These constructs address two major areas of functionality:

- objects and operations
  - objects
  - operations
  - invocation of operations
  - capabilities
  - etc.
- thread attributes
  - time constraints
  - exception handlers
  - etc.

The first area is discussed in Section 3.1.1, the second in Section 3.1.2.

3.1.1 Objects and Operations

This section describes the AOL constructs that define object types, specify operations on objects and their parameters, declare capabilities, and invoke operations on objects.

Throughout this section, two example object definitions are used to illustrate the various constructs. The `Stack` object definition consists of a naive implementation of an integer stack. The `Example` object definition performs operations on an object of type `Stack`. The complete object definitions are shown in Figures 13 and 14 in Section 3.1.1.6.

3.1.1.1 Object Definition

An object definition defines a particular type of object, of which there may be many *instances* active in the system at any time. (Analogously, in process-based systems the code for a program defines a type of process, of which there may be many instances running.) Figure 2 specifies the syntax of the object definition. Figure 3 shows an example of the object definition construct, with the contents elided.

Alpha Release 10 supports three kinds of objects: client, kernel, and system. A client object is specified by the **CLIENT** modifier, or the absence of a modifier. The client object is the primary type of object supported by Alpha; in this document, any general reference to "objects" should be taken to mean *client* objects, unless otherwise specified. Each client object has its own separate address space, in which threads execute at the user privilege level. Client objects have no access to the kernel except through the object invocation interface provided by the kernel.

$$\left[\begin{array}{c}\textbf{CLIENT}\\ \textbf{KERNEL}\\ \textbf{SYSTEM}\end{array}\right] \textbf{OBJECT}\ \textit{object\_name}\ \{\ \left\{\begin{array}{c}\textit{use\_statement}\\ \textit{capability\_declaration}\\ \textit{operation\_definition}\end{array}\right\}\ \}$$

Figure 2: Object Definition

```
CLIENT OBJECT Stack {
        •
        •
        •
}
```

Figure 3: Object Definition Construct Example

A kernel object definition (specified by the **KERNEL** modifier) is coded in substantially the same manner as a client object definition. A kernel object resides within the kernel and shares the kernel's address space (and C language global name space). Kernel objects offer a way to extend the object structure to parts of the kernel's functionality. Also, objects (and threads) that reside within the kernel incur less overhead. However, due to their considerable limitations and the degree of care required to write code that can exist innocuously within the kernel, the user is encouraged to employ client objects wherever possible.

A system object (specified by the **SYSTEM** modifier) represents a set of related kernel services provided to the user via an object interface. Since the functionality represented by system objects is implemented as standard C code inside the kernel, the system object definition contains no executable code. It is used to specify the interface to the system services, by specifying the operations with their formal parameter lists (see Section 3.1.1.2).

The *object name* is used to refer to the object type when instantiating it, declaring capabilities for it, etc. Note that an object name must be globally unique. Client objects can invoke operations on kernel objects, and *vice versa*, provided the invoking object has a capability for the object being invoked. Both client and kernel objects can invoke operations on system objects.

The body of a client or kernel object may contain anything that is legal in an ordinary C source file, plus **use** statements, capability declarations, and operation definitions. The **use** statement relates principally to system configuration and will be discussed in Section 3.2.1.2. Capabilities are used in operation invocation, and will be discussed in Section 3.1.1.3.

3.1.1.2 Operation Definition

The operation definitions in an object specify the legal user-defined operations on the object, with their parameters, and contain the code that implements the operations. Figure 4 specifies the syntax of an operation definition. Figure 5 shows the Stack object definition with operation definition constructs. (The data declarations and operation bodies are elided).

OPERATION *operation_name* **(** *formal_parameter_list* **) {** $\left\{ \begin{array}{c} \textit{use\_statement} \\ \textit{capability\_declaration} \\ \textit{invoke\_statement} \\ \textit{return\_statement} \\ \textit{deadline\_block} \\ \textit{exception\_block} \end{array} \right\}$ **}**

Figure 4: Operation Definition

The *operation name* is used to refer to the operation in invocations. It must be unique within the object definition of which it is a part. The formal parameter list is a comma-separated list of the formal parameters of the operation, with their respective types. Figure 6 specifies the syntax of a formal parameter.

Invocation parameters are passed by value or value-result. An **in** parameter is passed by value from the invoking object to the invoked object upon invocation. An **out** parameter is passed by value from the invoked object to the invoking object upon return from the invoked operation. An **in out** parameter is treated as both an **in** and an **out** parameter. Every formal parameter must be designated as an **in**, **out**, or **in out** parameter as shown in Figure 6.

The type of the formal parameter must be specified also. If the parameter is data, the type is specified just as any other type is specified in C. Note that, because parameters are passed by value across address spaces, it is generally not useful to pass pointers. If the parameter is a capability, its type is specified using the **CAPA** keyword. It is almost always a good idea to specify the type of object to which the capability refers. This allows more precise type checking on invocations involving the capability as either a parameter or as the object of the invocation (see Section 3.1.1.3).

```
CLIENT OBJECT Stack {
            •
            •
            •
    OPERATION Init() {
            •
            •
            •
    }
    OPERATION Push(IN int item) {
            •
            •
            •
    }
    OPERATION Pop(OUT int item) {
            •
            •
            •
    }
    OPERATION Exchange(IN OUT int topOfStack) {
            •
            •
            •
    }
    OPERATION Exchange2(IN int newTopOfStack, OUT int oldTopOfStack) {
            •
            •
            •
    }
}
```

Figure 5: Operation Definition Construct Examples

:

$$\begin{Bmatrix} \textbf{IN} \\ \textbf{OUT} \\ \textbf{IN OUT} \end{Bmatrix} \begin{Bmatrix} \textit{C\_type\_specification} \\ \textbf{CAPA} \begin{bmatrix} \textit{object\_type\_name} \end{bmatrix} \end{Bmatrix} \textit{name}$$

Figure 6: Formal Parameter

Generally speaking, the body of an operation may contain anything that is legal in the body of a C function. In addition, it may contain any of the AOL constructs specified in Figure 4. The **use** statement is discussed in Section 3.2.1.2, and the deadline block and exception block are discussed in Section 3.1.2. We discuss the capability declaration, **invoke** statement, and **return** statement below.

3.1.1.3 Capability Declaration

The syntax of the capability declaration is specified in Figure 7. (Examples of the capability declaration appear in the Example object definition in Figure 14, Section 3.1.1.6.) A capability declaration declares a named variable of type **capa.**

Recall from our earlier discussion of the Alpha programming model that a capability is a reference to an object, or a "handle" for referring to or accessing an object, via invocation. The Alpha Release 1 kernel interface treats capabilities as **unsigned long** values. (Actually, this is an index that is passed between an object and the kernel, which the kernel translates into an internal reference to the capability.) In terms of the AOL language extensions, the **capa** type, denoted by the keyword **CAPA**, should be treated as a built-in type that can be straightforwardly coerced to **unsigned long**, and *vice versa*.

[**WELLKNOWN**] **CAPA** [*object_type_name*] *capability_name* **;**

Figure 7: Capability Declaration

In one respect, the **capa** type is not treated exactly as a built in type: it is impossible to declare an array of **capa**. This restriction is made for consistency with well-known capabilities. A well-known capability (declared using the **WELLKNOWN** modifier) is a capability provided to an object automatically upon instantiation (as opposed to being passed in as an invocation parameter, the only other method by which an object can obtain a capability). The configuration system does not support arrays of well-known capabilities. Well-known capabilities are explained in more detail in section 3.2.2. For the purposes of this section, they may be considered identical to ordinary capabilities. Capabilities can be stored in an array of **unsigned long** if need be.

The capability declaration also associates an object type, specified by the *object type name* (the name given to the object type in its object definition), with the **capa** variable. This is the type of object to which the capability refers, and it should be specified wherever possible. A capability cannot be used as the object of an invocation unless its object type is declared.

A capability declaration may appear anywhere an ordinary data declaration may appear. The scope and extent of variables of type **capa** are determined by the same rules that apply to other variables in C. A **capa** may be an automatic variable or a global variable (there is no **static capa** declaration). A **capa** variable is, in fact, simply a variable to hold the index of a capability possessed by the object. In Release 1 of Alpha, capabilities are associated with an object; therefore it usually makes sense to declare **capa** variables as global.

3.1.1.4 Invoke Statement

The **invoke** statement allows a thread executing within an object to invoke an operation on another object, for which the invoking object has a capability. The syntax of the **invoke** statement is specified in Figure 8. (Examples of the **invoke** statement appear in the Example object definition in Figure 14, Section 3.1.1.6.)

[**UNCHECKED**] **INVOKE** *capability_name* **.** *operation_name* **(** *actual_parameter_list* **)**

{ **;** | *failure_clause* }

Figure 8: Invoke Statement

The *capability name* is the name of a declared **capa** variable that contains the index of the capability for the object to be invoked. The declaration of the **capa** variable must specify the type of the object to which the capability refers. The *operation name* is the name of the operation to be invoked. The *actual parameter list* is a comma-separated list of the arguments of the invocation. The syntax of an actual parameter is specified in Figure 9.

[**IN** | **OUT** | **IN OUT**] [**(** { *C_type_specification* | **CAPA** [*object_type_name*] } **)**] *C_expression*

Figure 9: Actual Parameter

An actual parameter consists of a C expression that evaluates to the value to be passed, and optional type information. The type of an actual parameter is specified with a syntax similar to that of a type cast in C. Whether the parameter is **in**, **out**, or **in out** may also be specified. The language translator will check any information provided against the interface specification of the object type being invoked. (The current implementation requires that type information be specified as part of the actual parameter in order for type checking to occur; the type-checking is based on lexical type name comparisons).

The concept of an interface specification is important in AOL. An interface specification for an object type consists of the operations it implements, along with their formal parameters. The utility of this notion is illustrated in the following example.

Suppose that, in the usual top-down fashion, we are coding an object definition that invokes objects of type Stack, but that the Stack object definition has not yet been coded. In order to invoke objects of type Stack, we must know what the interface is. Assuming we do know the interface, we can write an interface specification, such as that shown in Figure 10, for the Stack object type.

Note that this code is a legal AOL object definition. It provides the information necessary to type-check the **invoke** statements in the object definition we are coding, in a form that the language translator can parse. Later, the code that implements the Stack object type can be added to its definition. Both object definitions can be compiled separately, and they will be compatible as long as the interface specification for the Stack object type does not change.

```
CLIENT OBJECT Stack {
    OPERATION Init() { }
    OPERATION Push(IN int item) { }
    OPERATION Pop(OUT int item) { }
    OPERATION Exchange(IN OUT int topOfStack) { }
    OPERATION Exchange2(IN int newTopOfStack, OUT int oldTopOfStack) { }
}
```

Figure 10: Interface Specification for Object Type "Stack"

AOL requires that the interface specification be available for any object type corresponding to the type of the capability specified as the target of an **invoke** statement. However, by using the **UNCHECKED** modifier of the **invoke** statement, the user can disable type checking. Use of the unchecked invoke is to be discouraged, but it is sometimes necessary, for example in cases where the argument list needs to be variable.

The **invoke** statement is analogous to the function call in C. It is essentially a procedure call across protection and address space boundaries and, in a distributed system, if the invoked object resides on a different node than the invoking object, it is a remote procedure call. In Alpha, operation invocation is synchronous; the thread performing the invocation does not execute any further in the invoking object until the operation it has invoked completes.

Upon return from the invocation, two variables that indicate the status of the invocation are updated. These variables are accessible to the programmer using the special **RESULT** and **STATUS** variables defined in AOL. The **RESULT** variable indicates whether or not the invocation succeeded and, if not, whether the failure occurred in the kernel, or in the invoked object. **RESULT** is an unsigned short variable that has three possible values: 0 (**SUCCESS**), 1 (**FAILURE**), or 2 (**SYSTEM_FAILURE**). The **STATUS** variable is also an **unsigned short** which is set to the value specified by the invoked operation in its **return** statement (see Section 3.1.1.5). If the **RESULT** is **FAILURE** or **SYSTEM_FAILURE**, no **out** parameters are returned.

An **invoke** statement may end in a semicolon, or in a *failure clause*, the syntax of which is specified in Figure 11. The failure clause is a convenience to the programmer. The code within the body of the failure clause is executed if the **RESULT** returned from the invoke is **FAILURE** or **SYSTEM_FAILURE**.

$$\textbf{ON FAILURE \{} \left\{ \begin{array}{c} \textit{use\_statement} \\ \textit{capability\_declaration} \\ \textit{invoke\_statement} \\ \textit{return\_statement} \\ \textit{deadline\_block} \\ \textit{exception\_block} \end{array} \right\} \textbf{\}}$$

Figure 11: Failure Clause

3.1.1.5 Return Statement

The **return** statement causes control to return from the current operation to the invoking object. It is analogous to the **return** statement in C. The syntax of the **return** statement is specified in Figure 12.

$$\textbf{RETURN} \quad \begin{Bmatrix}\textbf{SUCCESS}\\ \textbf{FAILURE}\end{Bmatrix} \left[\textbf{(}\ \mathit{C\_expression}\ \textbf{)}\right]\textbf{;}$$

Figure 12: Return Statement

The **return** statement must specify whether the operation succeeded or failed. This determines whether the value of the **RESULT** variable is **SUCCESS** or **FAILURE** upon completion of the **invoke** statement. It may optionally return an **unsigned short** value (to which the expression in parentheses evaluates) that conveys more information (e.g., the reason for the failure). This determines the value of the **STATUS** variable upon completion of the **invoke** statement. The documentation of an object's interface should define the possible **STATUS** values and their meanings. A **return** statement is not required by the language; exiting at the end of the operation body ("falling through") is equivalent to returning **SUCCESS** with a **STATUS** of 0.

3.1.1.6 Examples

The example Stack object definition in Figure 13 serves to illustrate some aspects of the AOL extensions. It defines a client object type such that each instance of this type implements an integer stack. The integer array representing the stack and the integer stack pointer are global variables; they are stored in the object's (i.e., the object instance's) data area. In the Exchange operation the automatic integer variable temp is local to the thread executing the operation, and is allocated on the thread's client stack. This implementation of a stack does no synchronization, so only one thread at a time should be active in a Stack object. (In Release 1 of Alpha the kernel provides semaphores as system objects so that synchronization may be accomplished using P and V operations on semaphores.

The Example object type definition in Figure 14 contains some example **invoke** statements. Example contains only two operations, Run and Run2. Run uses a well-known capability, StackCapa, to invoke an object of type Stack. First, it initializes the stack by invoking the Init operation. The invocations of the Push operation illustrate the use of a constant as an **in** parameter, and the various amounts of type information that may be included in an actual parameter. These three invocations should have identical effects (i.e., pushing the value 3 on the stack). The invocations of the Exchange operation are also equivalent; the Exchange2 operation illustrates multiple parameters and the fact that using an **in** and an **out** parameter can achieve the same effect as an **in out** parameter. The last invocation of the Exchange2 operation uses different variables for the **in** and **out** parameters. It also uses a failure clause; if the operation fails, the Run operation of Example return failure. The last invocation is a Pop, which leaves two items on the stack.

The Run2 operation takes a capability for an object of type Stack as an **in** parameter and returns an integer as an **out** parameter. It also declares a local **capa** variable, and assigns a copy of the input capability to it. Each **invoke** statement uses a different capa variable to access the same Stack object.

```
CLIENT OBJECT Stack {

    int stack[MAX_STACK_SIZE];        /* this data is global within the object */
    int stackPointer;

    OPERATION Init() {
        stackPointer = -1;
    }

    OPERATION Push(IN int item) {
        if ((stackPointer + 1) > (MAX_STACK_SIZE - 1)) {
            RETURN FAILURE(ERROR_STACK_FULL);
        }
        stackPointer = stackPointer + 1;
        stack[stackPointer] = item;
    }

    OPERATION Pop(OUT int item) {
        if (stackPointer < 0) {
            RETURN FAILURE(ERROR_STACK_EMPTY);
        }
        item = stack[stackPointer];
        stackPointer = stackPointer - 1;
    }

    OPERATION Exchange(IN OUT int topOfStack) {
        int temp;            /* this data is local to the thread executing this operation */

        if (stackPointer < 0) {
            RETURN FAILURE(ERROR_STACK_EMPTY);
        }
        temp = stack[stackPointer];
        stack[stackPointer] = topOfStack;
        topOfStack = temp;
    }

    OPERATION Exchange2(IN int newTopOfStack, OUT int oldTopOfStack) {
        if (stackPointer < 0) {
            RETURN FAILURE(ERROR_STACK_EMPTY);
        }
        oldTopOfStack = stack[stackPointer];
        stack[stackPointer] = newTopOfStack;
        RETURN SUCCESS;
    }
}
```

Figure 13: Stack Object

```
KERNEL OBJECT Example {

    WELLKNOWN CAPA Stack StackCapa;

    OPERATION Run() {
        int item, item2;

        INVOKE StackCapa.Init();
        item = 3;

        /* the following are equivalent */
        INVOKE StackCapa.Push(3);
        INVOKE StackCapa.Push(IN item);
        INVOKE StackCapa.Push(IN (int)item);

        /* as are the following */
        INVOKE StackCapa.Exchange(item);
        INVOKE StackCapa.Exchange(IN OUT (int)item);
        INVOKE StackCapa.Exchange2(item, item);

        /* invoke with failure clause */
        INVOKE StackCapa.Exchange2(item, item2)
        ON FAILURE {
            RETURN FAILURE;
        }
        INVOKE StackCapa.Pop(item);
    }

    OPERATION Run2(IN CAPA Stack StackCapa2, OUT int item) {
        CAPA Stack Capa;

        Capa = StackCapa2;
        INVOKE StackCapa2.Push(IN (int)3);
        INVOKE Capa.Pop(item);
        RETURN SUCCESS;
    }
}
```

Figure 14: Example Object

3.1.2 Thread Attributes

AOL provides language constructs that allow the application programmer to specify the following thread attributes:

- time constraints (deadline block, Section 3.1.2.1)
- exception handlers (exception block, Section 3.1.2.2)

Details of thread exception processing are discussed in Section 3.1.2.3.

3.1.2.1 Deadline Block

The deadline block is used to specify scheduling constraints over an interval of thread execution. Its syntax is specified in Figure 15. The keyword **DEADLINE** is a misnomer, chosen because it is shorter than **TIME-CONSTRAINT** and because time constraints often take the form of a deadline (i.e., a hard time constraint requiring that a computation be complete by a specified time, and that it have no value if completed after that time).

$$\textbf{DEADLINE (}\ \textit{deadline\_arguments}\ \textbf{) \{}\ \left\{ \begin{array}{c} \textit{use\_statement} \\ \textit{capability\_declaration} \\ \textit{invoke\_statement} \\ \textit{return\_statement} \\ \textit{deadline\_block} \\ \textit{exception\_block} \end{array} \right\} \textbf{\}}\ \left[\textit{exception\_clause} \right]$$

Figure 15: Deadline Block

The deadline block is unique among AOL constructs in that it requires a set of highly specialized parameters with complex semantics. In order to describe these parameters, we must explain the Alpha scheduling interface. The goal of this discussion is to leave no doubt as to the significance of the specific parameters of the deadline block in the framework of the scheduling model implemented in Alpha Release 1. It is not the goal of this discussion to make the reader an expert in using the Alpha scheduler; that currently requires a detailed understanding of the workings of the scheduler and some experience. See Section 2.1.1.2 for a brief description of scheduling in Alpha. A more detailed discussion can be found in [Northcutt 88b].

Figure 16 shows how time-value functions for the best-effort scheduler of Alpha Release 1 are defined in terms of their parameters. Based on the value functions representing the tightest time constraint for each thread, the scheduler constructs a schedule intended to maximize the value of the computations that can be completed; it then determines which thread should be executing according to that schedule.

I	—	thread importance
t_c	—	critical time
t_e	—	expected execution time
$\sigma(t_e)$	—	standard deviation of t_e
S_{pre}	—	shape of value function, pre-critical time
S_{post}	—	shape of value function, post-critical time
i_{pre}	—	importance modifier, pre-critical time
i_{post}	—	importance modifier, post-critical time

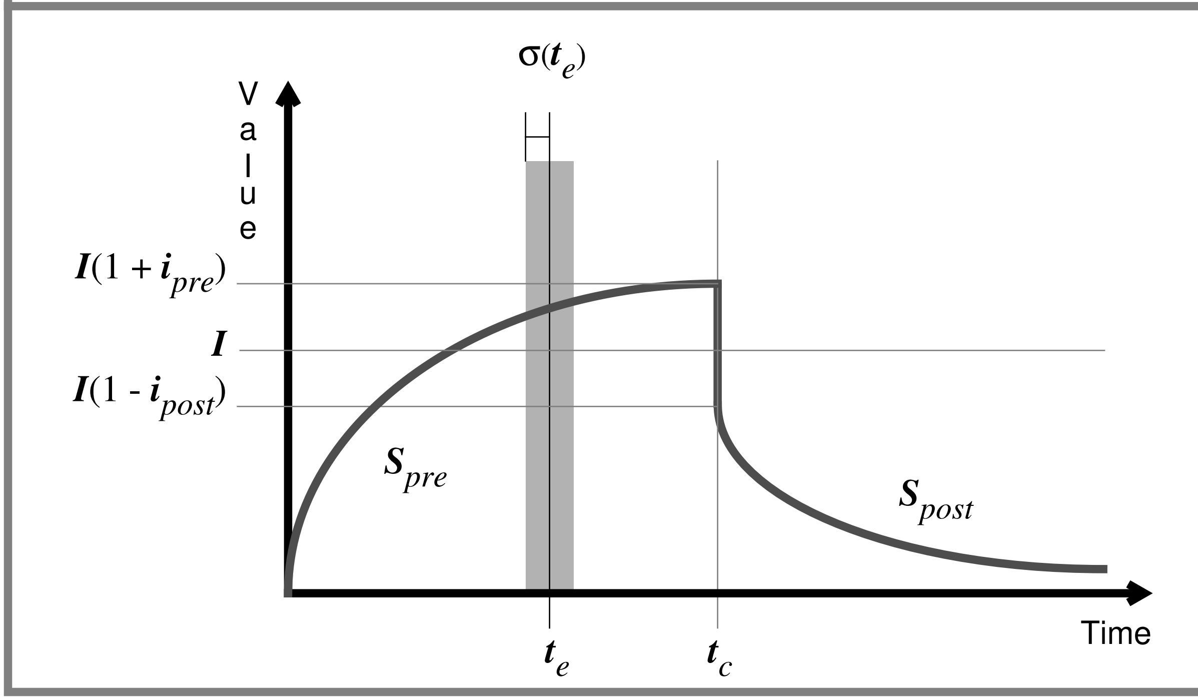

Figure 16: Time-Value Function Specification

The deadline arguments referred to in the deadline block syntax are simply the following time-value function parameters, in the order given: $\mathbf{t}_c$, $\mathbf{t}_e$, $\sigma(\mathbf{t}_e)$, $\mathbf{S}_{pre}$, $\mathbf{S}_{post}$, $\mathbf{i}_{pre}$, $\mathbf{i}_{post}$. The quantities $\mathbf{t}_c$, $\mathbf{t}_e$, and $\sigma(\mathbf{t}_e)$ are given in milliseconds, with type **float**. $\mathbf{S}_{pre}$ and $\mathbf{S}_{post}$ are indices of system-defined value-function shapes, of type **unsigned short**. The shapes of value functions currently defined in Release 1 of Alpha are illustrated in Figures 17 and 18. (An example of pre- and post-critical time value function types is shown in Figure 19.) The quantities $\mathbf{i}_{pre}$ and $\mathbf{i}_{post}$ are expressed as percentages, of type **short**.

The importance (**I**) of a thread is assigned based on the application. It is modified by invoking an operation on a system service object. For details see [Northcutt 88e].

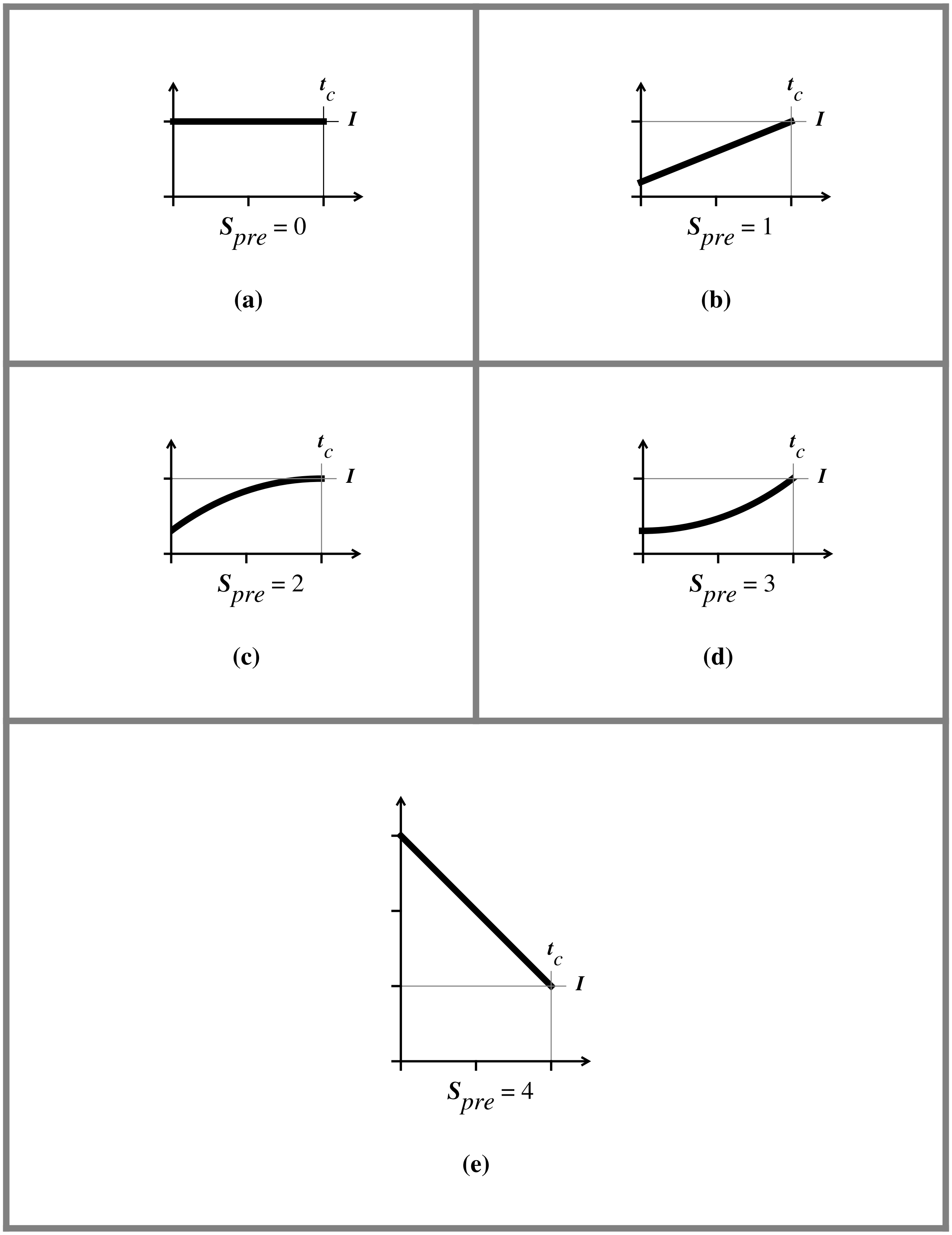

Figure 17: Pre-Critical Time Value Function Types

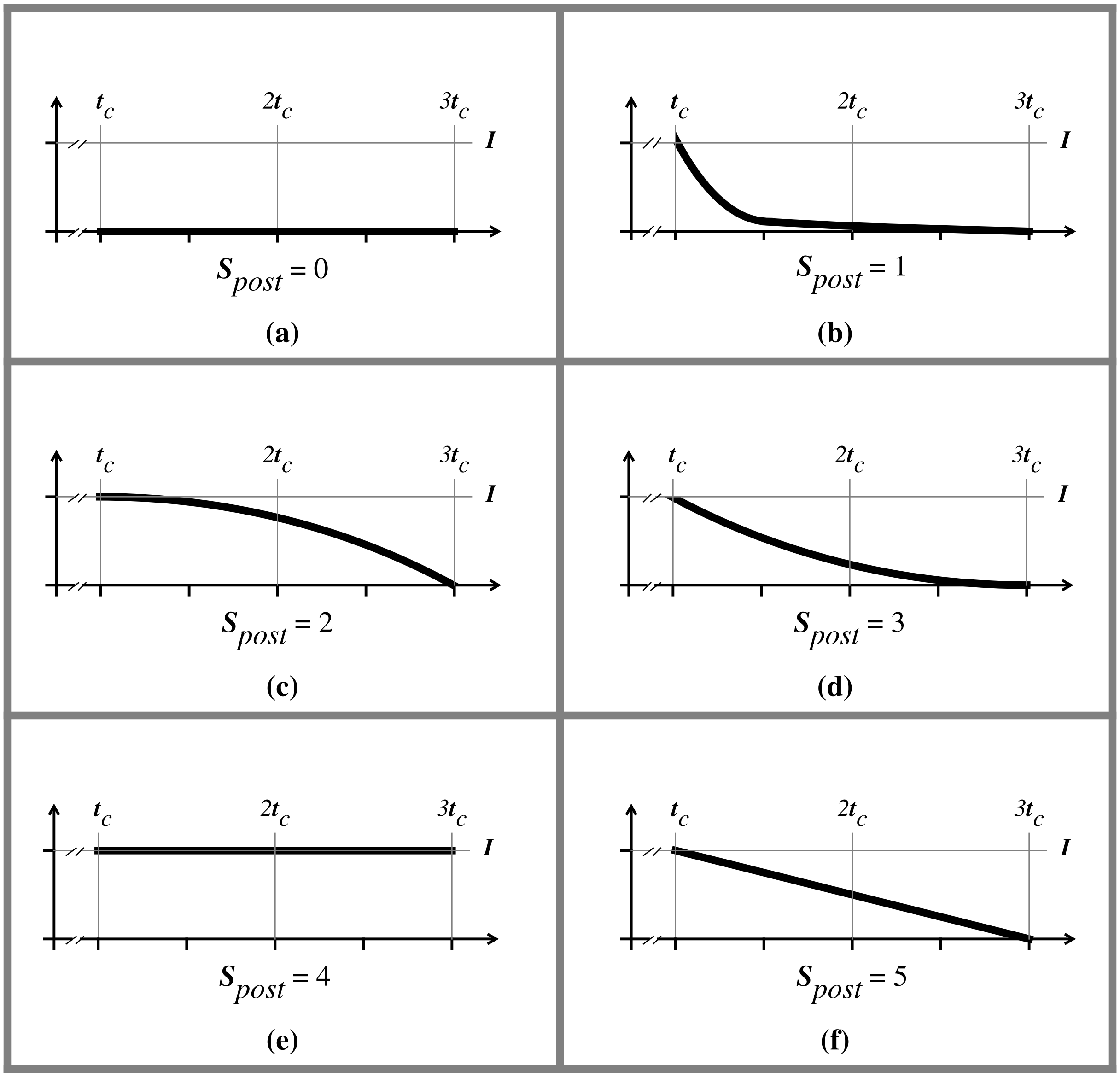

Figure 18: Post-Critical Time Value Function Types

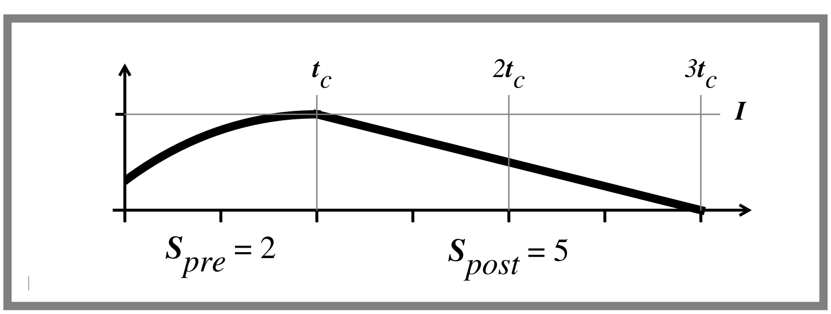

Figure 19: Example of Combined Pre- and Post-Critical Time Value Functions

While the thread is executing the code within the body of the deadline block, it will be scheduled according to the value function specified by the arguments provided. In the normal case, the thread will execute the code within the body of the deadline construct. Note that this code can invoke operations on other objects and can contain nested deadline blocks. (The result of transferring control directly into or out of a deadline construct, e.g., using a **goto** statement, is undefined). When control reaches the closing brace of the body of the deadline block, it exits the deadline block, and execution continues in the code following the end of the deadline block, which is scheduled based on the value function in effect before the deadline block was entered.

If, however, before control leaves the body of the deadline block, the value of the value function representing the tightest time constraint on the thread should fall to within *epsilon* of zero (*epsilon* being a small value determined by the scheduling policy), the thread is considered to have failed to meet its time constraint, and it experiences an *exception*. Exception handling and the *exception clause* are discussed in Section 3.1.2.3.

3.1.2.2 Exception Block

The exception block allows the application programmer to specify an exception handler for a section of code. The syntax of the exception block is given in Figure 20. Exception handling and the *exception clause* are discussed in Section 3.1.2.3.

$$\textbf{BLOCK \{} \quad \left\{ \begin{array}{c} \textit{use\_statement} \\ \textit{capability\_declaration} \\ \textit{invoke\_statement} \\ \textit{return\_statement} \\ \textit{deadline\_block} \\ \textit{exception\_block} \end{array} \right\} \textbf{\}} \left[\textit{exception\_clause} \right]$$

Figure 20: Exception Block

3.1.2.3 Exception Handling

The application writer specifies an exception handler for a section of code by using the exception clause of the deadline block (to specify an exception handler for the code within the body of the deadline block), or of the exception block (to specify an exception handler for the code within the body of the exception block). The syntax of the exception clause is given in Figure 21.

$$\textbf{ON EXCEPTION \{} \quad \left\{ \begin{array}{c} \textit{use\_statement} \\ \textit{capability\_declaration} \\ \textit{invoke\_statement} \\ \textit{return\_statement} \end{array} \right\} \textbf{\}}$$

Figure 21: Exception Clause

The block-structured deadline and exception blocks can be lexically nested within an object and dynamically nested across objects. Each of these constructs has its own exception handler associated with it, at its particular level of nesting. (If no exception clause is specified by the user, the system supplies a default exception handler that performs functions of importance to the system, such as maintaining the consistency of synchronization data structures). A particular exception occurs for some reason associated with a particular level of the nesting hierarchy. For example, a thread fails to meet a time constraint established at a particular level; a thread is broken at a particular level; a machine error occurs at a particular level (i.e., the current level). Thus we can associate a particular level with an exception.

When an exception occurs, control is transferred to the currently active exception handler (i.e., to the beginning of the body of the exception clause of the most deeply nested deadline or block construct). Machine registers contain the values they contained when the construct was originally entered. Local, stack-allocated variables accessible when the construct was entered are now accessible, and their values are unchanged (i.e., they have the same values they had when the exception occurred). (The exception handler could use the values of these variables to help determine how far the computation had progressed before the exception occurred.)

While executing the exception handler, the thread is scheduled according to the value function that was in effect when the construct was entered. Note that the exception handler cannot establish new exception handlers; also, invoking any operation that attempts to establish new exception handlers is considered an error. Generally, an exception handler is intended to be used by the application to clean up intermediate results and inconsistencies created by the asynchronous transfer of control. Exception handlers should not require a great amount of execution time. As in the case of the body of a deadline or block construct, the result of jumping into or out of an exception clause is undefined.

When control leaves the exception clause, what happens depends on whether the exception is associated with the current level, or with a higher level. If the exception is to the current level, the exception is no longer active, and execution continues normally with the code following the exception clause. If the exception is associated with a higher level, then control is again asynchronously transferred to the exception handler associated with the next higher level of nesting. This process continues until the level with which the exception is associated is reached.

Exception processing proceeds sequentially within a node, as described above, but can proceed in parallel among nodes. If a thread exception is at a level such that the sections of the thread at lower levels reside on multiple nodes, each node performs exception processing in parallel. Since exception processing is inherently local to a node (except for coordination), and the exception handlers are local to objects, this poses no difficulty to the application programmer.

More information on exception processing may be found in [Northcutt 88b].

3.1.2.4 Example

An example of the use of the deadline block and exception clause, taken from an application written by General Dynamics personnel, is shown in Figure 22. The application is described in detail in [Clark 88].

```
/* Main guidance loop containing the most important deadlines */
while (still_possible) {

    DEADLINE (delay_time, 1000, 100, 0, 0, 25, 10) {
        INVOKE AI_sleep.Psem();
    } ON EXCEPTION {
        USEREVENT(23, 0x4f);
    }

    DEADLINE (deadline_time, 1000, 100, 0, 0, pre_dln_imp, post_dln_imp) {
        INVOKE TrackDB.GetTrackInfo( IN (id_type) ai_id,
                                     OUT (track_type) ai_track);
        INVOKE TrackDB.GetTrackInfo( IN (id_type) hostile_id,
                                     OUT (track_type) hostile_track);
        if ((hostile_track_allocated == FALSE) && (ai_track_allocated == FALSE) {
            still_possible = FALSE; hostile_dead = friendly_dead = TRUE;
        } else if (ai_track.allocated == FALSE) {
            still_possible = FALSE; friendly_dead = TRUE;
        } else if (hostile_track.allocated == FALSE) {
            still_possible = FALSE; hostile_dead = TRUE;
        } else {
            /* guidance calculations (code deleted) */
            /* create SIM report (code deleted) */
            INVOKE Comm.SendMsg(     IN (msg_t) ALPHA_RPT_GUIDE,
                                     IN (u_int) size,
                                     IN (rpt_guide_t) rpt_guide);
        }
        /* set up next deadline */
        INVOKE ThreatAssess.AssessThreat(  IN (id_type) hostile_id,
                                           IN (track_type) hostile_track,
                                           OUT (int) threat_priority);
        pre_dln_imp = threat_priority * 10;
        post_dln_imp = threat_priority * 10;

        /* make Airborne Interceptor 15% faster than the hostile (code deleted) */
        delay_time = (time_to_int / 2) * 1000;       /* convert to microseconds */
        deadline_time = delay_time / 2;
    }
}
```

Figure 22: Deadline Example

This is a part of the application that is guiding an airborne interceptor to intercept a hostile missile. The loop updates the guidance of the interceptor in a loosely periodic fashion. The first deadline block causes the thread to block for a specified period of time. In most operating systems, a special system call would be used to achieve this effect, but in Alpha the combination of sleeping on a semaphore (to block the thread) and taking an exception when the deadline expires (to wake the thread up) accomplishes this function through composition of system primitives. The exception clause simply logs an event for later analysis of the system's behavior. In the next deadline block, the actual guidance is performed. (If the time constraint is not met, the course correction will not occur; if this happens repeatedly, the interceptor gets further and further off course). The application gets track information about the interceptor and the hostile missile. It checks to see whether any of the tracks have disappeared (indicating a significant event). Otherwise, it sends a message to the simulator with its course correction for the interceptor. Then, it dynamically computes the importance, based on the assessed threat potential of the hostile, for the next iteration of the guidance loop.

3.2 Configuration Constructs

The configuration definition is the top-level AOL construct that embodies the programming language support for system configuration. This support takes the form of special AOL constructs for specifying initial configurations of objects and threads on nodes, and a kernel-provided abstraction, also supported in AOL, called a *well-known capability*, for specifying initial object access permissions. We discuss object and thread initialization constructs in Section 3.2.1, and well-known capabilities in Section 3.2.2.

3.2.1 Configurations

This section describes the AOL constructs that define system configurations, node configurations, and initial object instances and threads. The structure of configuration definitions and node definitions parallels that of object definitions and operation definitions. Some examples of configuration constructs are given in Section 3.2.2.2.

3.2.1.1 Configuration Definition

The configuration definition allows the user to define a configuration and give it a name. It defines a configuration for an entire system consisting of many nodes. The syntax of a configuration definition is given in Figure 23.

$$\textbf{CONFIGURATION}\ \textit{configuration\_name}\ \mathbf{\{}\ \left\{\begin{array}{l} \textit{use\_statement} \\ \textit{map\_statement} \\ \textit{node\_definition} \end{array}\right\}\ \mathbf{\}}$$

Figure 23: Configuration Definition

The configuration definition may contain *use statements*, *map statements*, and *node definitions*. The **map** statement will be discussed in conjunction with well-known capabilities, in Section 3.2.2.1.

3.2.1.2 Use Statement

The **use** statement is of dubious utility, but was allowed to remain in the language in case it is ever needed. It allows the user to specify object types that should be part of the configuration. A **use** statement is not needed unless the object type in question was never referenced by any other object (using AOL constructs). It was used at one time to include a kernel object that was invoked by C code within the kernel, but was never the target of an AOL **invoke** statement. However, since that object became a permanent part of the kernel, the configuration software was changed to include it automatically in all configurations. This example indicates how rarely the construct is used in practice. Its syntax is given in Figure 24.

USE *object_type_name* [{ **,** *object_type_name* }] **;**

Figure 24: Use Statement

3.2.1.3 Node Definition

The node definition specifies the configuration of specific nodes of the system. It was originally intended that there be a node definition for every node in the system. However, it was noted that in some cases, several nodes could use the same initial configuration. In this case, it makes more sense to view a node definition as defining a *type* of node. Thus, there should be a node definition for each different initial configuration of nodes used in the system. Each such configuration has a name, specified in the node definition. Its syntax is given in Figure 25.

NODE *node_name* **{** [{ *use_statement* | *instance_definition* }] **}**

Figure 25: Node Definition

The node definition may contain *use statements* and *instance definitions*.

3.2.1.4 Instance Definition

An instance definition causes an instance of an object type, specified in the **instance** statement, to be created at initialization. The instance is given a name, which may be used to refer to it in **map** statements. The syntax of the instance definition is given in Figure 26.

INSTANCE *instance_name* **OF** *object_type_name* **{** [{ *thread_statement* }] **}**

Figure 26: Instance Definition

The instance definition may contain *thread statements*.

3.2.1.5 Thread Statement

A **thread** statement causes a thread to be created during initialization, after the creation of the instance defined by the instance definition containing the **thread** statement. The thread initially invokes the specified operation on this object instance. Multiple **thread** statements are allowed because multiple threads can be active within an object. The syntax of the **thread** statement is given in Figure 27.

```
THREAD thread_name INVOKES operation_name ( actual_parameter_list ) ;
```

Figure 27: Thread Statement

Note that any actual parameters in the actual parameter list would have to be constants, since there is no provision in the configuration definition for declaring variables (or writing any other code).

3.2.2 Well-Known Capabilities

A well-known capability is a capability provided to an object instance when it is created—i.e., an object is "born with" well-known capabilities. An object type indicates that it is using a well-known capability by declaring the capability using the **WELLKNOWN** modifier in the capability declaration. Every instance of the object type will be created with the well-known capabilities declared in the object type definition. The names of well-known capabilities are global across the entire configuration (i.e., all well-known capabilities declared in all object types used in a particular configuration must have unique names).

Generally, the type of object referred to by a well-known capability must be known when the code using that capability is written, since invoking operations on an object requires knowing its interface. However, the particular instance being invoked is not known. Thus well-known capabilities provide a level of indirection in object referencing. The object instances to which the well-known capabilities refer must be defined at system configuration time.

3.2.2.1 Map Statement

This is done using the **map** statement in the configuration definition. The syntax of the **map** statement is given in Figure 28.

```
MAP capability_name TO instance_name ;
```

Figure 28: Map Statement

The **map** statement establishes the correspondence between the well-known capability name and the name of the object instance to which it should refer (generally, a name specified in an instance definition within one of the node definitions). In addition to any well-known capabilities that the application may map, the system automatically maps the names of the system objects to corresponding well-known capability names. This allows an object to access the system services of the node on which it resides using a well-known capability. Also, the system automatically maps the names of object types to corresponding well-known capability names, so that any object can create an instance of any object type using a well-known capability for the object type.

Builders of applications concerned with fault tolerance should note that the indirection provided by well-known capabilities is available at system-build time only. For the entire life of the system, objects are created with their well-known capabilities being mapped to the capabilities of the object instances specified at system configuration. The system's internal facilities for creating and manipulating well-known capabilities are not available to the application. Thus, an invocation on a well-known capability might fail. For example, a node may crash, destroying an object instance that corresponded to a well-known capability. Therefore, applications attempting to tolerate node failures should check for failures of such invocations and deal with them appropriately.

3.2.2.2 Examples

We conclude this section with two examples of configuration definitions. The configuration definition of the Sample configuration, shown in Figure 29, combined with the Example object shown in Figure 14 and the Stack object of Figure 13, provides all the application code necessary to produce a configuration in which the Run operation of the Example object is executed.

```
CONFIGURATION Sample {
    MAP StackCapa TO StackIns;
    NODE I {
        INSTANCE ExampleIns OF Example {
            THREAD ExampleThr INVOKES Run();
        }
        INSTANCE StackIns OF Stack { }
    }
}
```

Figure 29: Sample Configuration

The second example is a configuration of the application written by General Dynamics personnel, named C2Demo, shown in Figure 30. This application, which is described in considerable detail in [Clark 88], communicates with a UNIX system running a simulator that provides the environment to which the real-time system, running under Alpha, responds (and which it manipulates via operations that send commands to the simulator). At the beginning of the configuration, several **map** statements set up initial connections among components of the application. In particular, the first **map** statement maps a well-known capability name IOLink to an instance of the object type NetIO. In earlier stages of development, we used a serial line to communicate with the UNIX system, instead of the Ethernet. There is an object type SerialIO that handles communication via the serial line, just as NetIO handles communication via the Ethernet. Both objects have the same interface, and it is possible to switch between them simply by changing the **map** statement in the configuration definition.

Also note that the definition of node type Interface defines the configuration for the single node that interfaces with the UNIX-based simulator. The node type Application defines the node configuration used for all other nodes in the system.

```
CONFIGURATION C2Demo {
    MAP IOLink TO NetIOIns;
    MAP Comm TO CommIntfIns;
    MAP DistrServ TO DistrServMgrIns;
    MAP TrackDB TO TrackDBMgrIns;
    MAP Weapons TO WeaponsMgrIns;

    /* Interface Processing Node */
    NODE Interface {
        INSTANCE CommProcIntfInsI OF CommProcIntf {
            THREAD CommProcIntfThrI INVOKES ServiceCP();
        }
        INSTANCE NetIOIns OF NetIO {}
        INSTANCE CommIntfIns OF CommIntf {}
        INSTANCE DistrServMgrIns OF DistrServMgr {}
        INSTANCE TrackDBMgrIns OF TrackDBMgr {}
        INSTANCE WeaponsMgrIns OF WeaponsMgr {}
        INSTANCE IntfInitIns OF IntfInit {
            THREAD IntfInitThr INVOKES Init();
        }
    }

    /* Application Processing Node */
    NODE Application {
        INSTANCE CommProcIntfInsA OF CommProcIntf {
            THREAD CommProcIntfThrA INVOKE ServiceCP();
        }
        INSTANCE ApplicationInitIns OF ApplicationInit {
            THREAD ApplicationInitThr INVOKES Init();
        }
    }
}
```

Figure 30: C2Demo Configuration

4 Rationale

This chapter provides the rationale for the language design presented in Chapter 3. It deals first with the overall language structure (Section 4.1), then with the details of the kernel interface constructs (Section 4.2) and the configuration constructs (Section 4.3).

The development of AOL was not strictly top-down. Implementation considerations had some effect on the design of the language, and that is indicated where relevant in the following discussion. Details of the actual implementation are given in Chapter 5.

4.1 Language Structure

The Alpha Release 1 programming language support consists of a set of extensions to the C programming language. C was chosen as a basis for many reasons. It is well-supported in the Alpha cross-development environment (SunOs) and well-known to a great many programmers, including the Alpha personnel. This provided great leverage, since it was not necessary to design and implement an entirely new language from scratch. Also, a set of appropriately designed extensions could be implemented with a preprocessor. This was felt to be a more expedient approach than modifying an existing C compiler.

Furthermore, C provides a good framework for the Alpha kernel mechanisms, since it provides none of their functionality itself. In particular, C is not object-oriented, has no built-in support for exception handling or concurrent programming, and does not address programming in the large (beyond separate compilation of code modules). This provides an environment in which the Alpha kernel mechanisms can be evaluated on their own merits. (How well they support other language paradigms, such as those found in Ada, is a matter for future research.)

The AOL language was designed to be upward-compatible with C, in the sense that a compiler for AOL could also accept standard C. If no top-level AOL construct (object or configuration definitions) is present in the input source file, the compiler should behave exactly like a standard C compiler. If there are top-level constructs in the file, the compiler should accept them and allow any AOL extensions within them.

In AOL, object and configuration definitions are language constructs, independent of files. In other words, a file can contain one or more object definitions or configuration definitions. This design was arrived at in imitation of Ada, and was considered to be more elegant than the direct source file to object file correspondence used in C, and more appropriate to an object-oriented language.

However, in the context of the C language and the UNIX development environment it now seems better to enforce a direct correspondence between a source file and an object (or configuration definition). An object definition results in a single C language source file. Application programmers typically write one object definition per file. And a less awkward way of mapping object type names into file name could be used. Currently, a user-defined *map* file, which contains the mapping from object type names to file names, is used. With file names corresponding directly to object type names, a *search path* (a UNIX-supported search list of file directories) could be used to find the source files. This requires less work on the part of the user, and conforms to standard UNIX practice.

The AOL constructs were designed to fit reasonably well into the C framework. Thus constructs such as the object definition, operation definition, deadline block, and exception block are block-structured and delimited by curly braces. An operation definition is similar to a C function definition, a capability declaration is similar to other C declarations, etc. However, the lexical representation of the language is designed to make it easily distinguishable from C code so that a preprocessor can easily process it. Thus every construct begins with a distinct keyword that is not part of the C language, sometimes only by virtue of C's case sensitivity, as with the **return** statement. AOL keywords are capitalized for this reason, and to make them stand out from the standard C code for greater readability.

4.2 Kernel Interface Constructs

The kernel interface constructs constitute a fairly direct mapping of kernel mechanisms (objects, operations, capabilities, invocation, thread attributes, exception handling) into language constructs (object definition, operation definition, capability declaration, **invoke** and **return** statements, deadline and exception blocks, exception clause).

The object definition allows the application programmer to define and name object types. The operation definition provides named operations with formal parameters of arbitrary type. The name of an operation can be used in **invoke** statements; at system build time it is automatically mapped into the actual operation index required by the kernel. The value-result nature of parameter passing in invocation is supported by providing for parameters to be declared as **in**put, **out**put, or both, with respect to the operation.

The capability declaration allows the application programmer to associate an object type with a capability, to support operation name mapping and parameter type checking for invocations. It resembles other C declarations, but does not provide their full generality (e.g., arrays of capabilities may not be declared) because its primary function is to provide the object type-capability association. Also, capabilities are treated differently than ordinary parameters in invocation parameter passing, and the language translator must be able to determine which parameters are capabilities in order to pass them correctly.

The **invoke** statement is a straightforward procedure call construct. Although object operations are similar to C functions (and are implemented as C functions with code to handle parameter passing, in the current implementation), the Alpha invocation semantics are different from C function call semantics (e.g., no distinguished return value, value-result parameter passing, the possibility of failure due to the distributed nature of the underlying hardware). The **invoke** statement better fits those semantics. It would be possible, though awkward, to make an **invoke** statement look like a C function call, but that would much more difficult for a preprocessor to identify and translate.

The **invoke** statement allows the complete specification of parameter types (**in/out** and data type). The language translator should perform type checking on any information provided. The data type may be provided explicitly because a preprocessor-based implementation might not be able to determine the data type of an expression; the C code can be generated so as to cause a C compiler error message if the provided type information is incorrect. The preprocessor can do type checking based on type names. The data type is specified by a construct similar to a type cast in C so that AOL code with parameter types specified could be upward compatible with some later version of the language that would not require the type to be specified as part of the parameter in order to do type checking.

An earlier version of the system allowed invocation without providing the object type, and depended on the specification of parameter types and the operation index (or used somewhat arbitrary defaults) in order to generate working code. This feature was discontinued because the object type information, and an interface specification for the object type, should always be available. Parameter type checking may be disabled using the **UNCHECKED** modifier. This is mainly useful for variable parameter lists. For example, the thread create operation on the ThreadManager system object takes the parameters for the thread's initial invocation as the trailing parameters of its invocation. (The ability to define operations with variable parameter lists is not provided to the user by AOL).

The use of interface specifications allows for completely separate compilation of objects. The final resolution of operation names and a final consistency check is done at system build time. Interface specifications are also useful for defining the invocation interface to system services (which are not written in AOL, but provide and object interface). Object types can be distributed without source code (binary only) if an interface specification is distributed with the binary. Also, using interface specifications allows object types to be developed completely independently (except for interface changes) until the system is ready to be tested.

The **RESULT** and **STATUS** variables provide straightforward access to the system-defined status of an invocation. The failure clause of the **invoke** statement is modeled on the exception clauses of deadline and exception blocks, and is provided as a programmer convenience. The **return** statement allows the application programmer to indicate the status of a complete operation. Originally, "falling through" an operation (exiting without encountering a **RETURN** statement) resulted in a failure return, which required the programmer to explicitly return **SUCCESS**. However, for reasons of esthetics, programmer convenience, and consistency with C, the default is now a successful return.

The deadline and exception blocks are lexically nested. They are implemented as block-structured constructs that must fit into the nesting hierarchy defined by the occurrence of other nested constructs in the language (e.g., compound statements). This enforces the dynamic nesting of thread attributes (Section 2.1.1.2), but is more restrictive than necessary. Our experience using the deadline block to specify scheduling constraints indicates that the required semantics may not fit this syntactic framework very well, and that it may be necessary to provide more constructs or relax the lexical nesting requirement in later versions of the system. However, this is fundamentally a kernel interface issue, rather than a language issue; it is discussed in [Northcutt 88e].

The exception clause is implemented as code at the same level of block nesting as the body of the deadline or exception block, rather than as a separate routine. This gives the exception handling code access to the same local stack-allocated variables as the block body. This makes it possible for the exception handling code to infer something about the progress made by the thread within the body of the construct based on the values of those variables. It also reflects the kernel's implementation of exception handling.

The deadline arguments are an awkward part of the deadline construct. Perhaps a better approach would be to have one argument, a system-defined **struct** type containing the deadline arguments. The current implementation does very little checking on the arguments in any case (however, the kernel does check them).

The exception clause does not allow nested deadline or exception blocks for obvious reasons. Pending exceptions are processed sequentially. Invocations are allowed, mainly for access to system services, but exception handling should be completed as rapidly as possible.

4.3 Configuration Constructs

The configuration constructs provided in AOL allow the application programmer to easily specify the initialization of an embedded system. (See discussion in Section 2.2.2). If the system is very static, the user can specify all the object instances, threads, and well-known capabilities (see below) that will exist in the system, and the system will automatically perform the initialization.

Alternatively, if the system is highly dynamic, the user can specify a single instance of an application-specific initialization object for each node, with a single thread to be started in its single initialization operation, that will perform all further system initialization. The user may choose not to use well-known capabilities (except for system service objects and object types in the initialization object) and distribute all capabilities via object invocation.

The AOL configuration constructs provide a straightforward way to specify named initial configurations of object instances and threads on nodes of the Alpha distributed system testbed. Note that the mapping of named node sub-configurations to actual testbed nodes is not specified in the language. This mapping can be determined at a later time (e.g., at system start-up). This allows configuration to adapt to the state of the testbed (e.g., different nodes may be available at different times). Also, the mapping may be one-to-many (i.e., multiple nodes may be initialized in the same way). This allows a single configuration to adapt to different numbers of nodes available, if the application supports that.

AOL deals with the problem of initial distribution of capabilities by using an Alpha kernel-supported mechanism, the well-known capability (which was added to Alpha in anticipation of this need). Well-known capabilities eliminate the need to pass many capabilities around as parameters to operations. This speeds up initialization and reduces the size of parameter lists. In the case of object types and system services, which are declared implicitly by the system, the user need not figure out exactly which system services and/or object types it uses and make sure their capabilities have been passed to it. Also, the configuration system automatically determine which object types should be included in the configuration, based on references in the configuration definition and the object definitions. This approach eliminates some unnecessary bookkeeping for the application programmer.

Well-known capabilities provide a level of indirection in object naming, but it is only available at system build time. This can result in the situation described in Section 3.2.2.1, in which, for example, a newly created instance of an object can be created with a well-known capability that does not work, because the object instance assigned to it at configuration time no longer exists, e.g., because of a node failure. No automatic method of dealing with this problem was implemented in Alpha because it was felt that such a policy should be an application-level decision. The indirection is still useful, because the user can substitute objects at system build time if their interface specifications are the same. For example, a communication package that uses the network can be substituted for one that uses a serial line simply by changing a **map** statement in the configuration definition, if they both present the same interface. (The configuration program could generate a warning if the type of the instance assigned to the well-known capability is different for the type given in its capability declaration(s).)

The configuration system does have some deficiencies that stem from the design of the configuration definition. Two observations lead to this conclusion. One is that users of the system tend to write their own “initialization objects” and use the configuration definition mainly to start up a thread in their custom initialization object, which proceeds to create objects and threads to implement the application. Ideally one would wish to combine this functionality with the configuration definition, so that a special user-defined initialization object is not needed. However, the configuration definition has no place for user-written C code.

This is also a problem in specifying the invocation parameters of a thread's initial invocation in a **thread** statement. Only constants can be specified, because there is no code to compute variable quantities. One possible alternative design would eliminate the configuration definition as a separate construct and have constructs similar to those now allowed in a configuration definition appear as extensions to ordinary AOL code (found in object definition) that could only be used in a distinguished Configuration object definition, perhaps with node constructs analogous to operation definitions. This would allow the user to customize the start-up code without foregoing the convenience of the configuration language extensions.

5 Implementation

The implementation of the Alpha Release 1 programming language support is interesting because it illustrates how the use of good implementation techniques can produce, in a few programmer-months, a close approximation to functionality that, if implemented in its entirety in the most obvious way, might require much greater effort.

The Alpha cross-development environment consists of a combination of standard UNIX software development facilities and custom Alpha-specific and general-purpose tools running under UNIX and Alpha and provides users with a clean, high-level interface for cross-developing Alpha applications under UNIX. This report concentrates on the tools that implement most of the programming language support:

- the AOL Preprocessor, occ (discussed in Section 5.2)
- the Alpha Configuration Program, acp (discussed in Section 5.3).

The structure of the implementation as a whole is discussed in Section 5.1. Other tools are described in [Northcutt 88d].

The current implementation was produced with approximately three programmer-months of effort (plus maintenance and enhancements as needed), and consists of 10,000 lines of C, lex, and yacc source. It met our requirement that the implementation consume relatively little effort, yet it delivers substantial functionality. However, the system seems to be straining the limits of applicability of the implementation techniques used, and any substantial enhancements would likely call for a new implementation.

5.1 Implementation Structure

A major design decision was to implement the language using a preprocessor rather than a compiler. Section 5.1.1 discusses the rationale for and implications of that choice, mainly in the context of the AOL preprocessor. Other implementation techniques used are described in Section 5.1.2.

5.1.1 Preprocessor vs. Compiler

For several reasons, the decision was made early on to implement the AOL extensions using a pre-processor. It was believed that a preprocessor-based implementation would require less effort than a compiler-based one (even modifications to an existing compiler), and that it would provide adequate functionality for a system that would be used mainly, though not exclusively, by its authors (at least initially). Also, we would be sure that we could distribute the source code for the preprocessor, whereas a modified compiler might be problematic in that regard. Since the major emphasis of our research is on operating systems rather than programming languages, the trade-offs were considered acceptable.

Technically, we realized from the outset that a compiler-based implementation could offer advantages. For example, type checking of invocation parameters in the preprocessor can only be accomplished if the programmer explicitly includes the types of the arguments in the actual parameter list. A compiler would know the types of the arguments.

In the current implementation, AOL constructs such as **invoke** statements can be used only in operation definitions, and, in particular, cannot be used in an ordinary C function. This is because the pointers to the special parameter passing area of the thread's address space, which are required to marshal parameters for an invocation, are kept on the stack (i.e., they are local to the thread) and therefore would have to be passed to the function somehow. A compiler based implementation could handle this automatically. Furthermore, a compiler could allocate parameters to operations directly in the special parameter area, thus eliminating the need for copying parameters into and out of this area.

A compiler-based implementation could also allow capabilities to be treated as a first-class type in C. Currently the only reason for declaring **capa** variables is to associate the object type with the capability. In a compiler-based implementation, a **capa** could be a field in a structure, or an element of an array. Also, a compiler could better enforce restrictions on AOL constructs (such as not allowing jumps into and out of deadline blocks, etc.)

In the current implementation, the preprocessor does not parse the C code fully. This is because parsing the entire C language would require much more code in the parser, especially since the lexical analysis of C is context sensitive, due to the presence of the **typedef** facility. This results in some deficiencies in the language accepted in AOL constructs. For example, C type specifications used in AOL constructs are restricted to consisting of a single identifier (this is not an important functional restriction, because any required type can be named using a **typedef** statement). Also, C expressions used in AOL constructs are restricted to being either constants or single identifiers. A compiler-based implementation would not have these problems.

From a utilitarian point of view, however, the full C syntax for these constructs could be allowed (without checking by AOL) by forcing the lexical analyzer to accept whatever text is between the tokens that delimit these constructs. However, this was not implemented because the lexical analyzer was already rather complex, and adding this feature would have made it even more so. Again, this is not a serious problem in practice since types or expressions can be easily **typedef**ed or assigned to a single identifier.

The lexical analyzer is complex because it operates in two modes. It passes through all C code, comments, etc., and only returns tokens for the AOL construct (and C constructs that the parser needs to be aware of for context, e.g., braces, which are needed to check for proper nesting). This required making sophisticated use of lex facilities in order to effect the mode changes at the proper points in the input. (The AOL keywords are all upper case to make them easily distinguishable from C keywords, both for readability and to simplify the lexical analysis).

The lexical analyzer is one areas in which, paradoxically, the use of what might loosely be termed a prototyping language might actually have been counter-productive. The complex nature of the lexical analysis resulted in considerable painstaking work to get lex to do the right thing. Although lexical analyzer consists of only a couple of pages of (sophisticated) lex code, writing it from scratch would probably have been easier in the long run, more maintainable, and would almost certainly have resulted in better performance. (The preprocessor's performance has not been perceived as a problem so far.)

Parsing only the AOL constructs led to an interesting design decision: to run the code through the AOL preprocessor (occ) *before* running it through the C pre-processor (cpp). This approach has a number of advantages. cpp can assist in the implementation (the keywords **RESULT**, **STATUS**, **SUCCESS**, **FAILURE**, and **SYSTEM_FAILURE** are implemented as cpp macros defined in the generated code). The user can use cpp to compensate for some deficiencies of the AOL implementation, such as by defining cpp macros for elaborate expressions and using the macro name where AOL requires an identifier (this is something of a hack, but effective). Also, the code produced by occ is fairly readable by a human (unlike code produced by cpp, in which odd spacing results from include files and macro expansions, and the semantic level is lowered by replacing macro names with their expansions). This is useful for debugging. Also, it is simpler for occ to generate the proper line number comments so that syntax errors found by the C compiler are reported based on the line numbers in the original (AOL) source.

There are also some drawbacks to this approach. A user could unknowingly define macros with names that correspond to strings in the code generated by occ. But our coding conventions (and the fact that we have friendly users) made this highly unlikely. Also, it is pointless to define cpp macros whose definitions contain AOL constructs, since they will not be processed by occ. We have so far had little use for such macros. Perhaps the most important drawback is the confusion of semantics that arises from the arrangement. The notion of translating directly from syntactically correct C (as opposed to C extended by cpp macros) augmented by AOL constructs into standard C seems more intellectually manageable. But, again, in practice this has caused no problems.

5.1.2 Other Implementation Techniques

None of the ideas in this section is particularly surprising or unique. However, the evident value in their application to this implementation effort serves as another illustration that using good software design techniques pays off.

The AOL parser was written using a grammar-based parser generator, yacc. It is a parser for the entire language, and the parser actions consist exclusively of calls to action routines, made via an indirection (jump) table. This table is initialized with the addresses of the appropriate action routines before it is called. This was a very effective technique for using the same parser for multiple purposes. (For example, when parsing object definitions, the routines placed in the jump table entry for handling configuration constructs would report an error condition.) This parser is used for both passes of occ, as well as for acp.

In addition to the AOL parser, the routines that implement the data structures representing a configuration (components of which include objects, instances, operations, formal parameters, threads, well-known capabilities, etc.) are shared between occ and acp.

The configuration data structures are built up from more primitive data structures, such as:

- queue, with the operations:
    - enqueue
    - dequeue
    - insert
    - delete
    - swap
    - empty-predicate
- set, with the operations:
    - clear
    - insert
    - remove
    - union
    - difference
    - complement
    - empty-predicate

- symbol table, with the operations:
    - insert
    - replace
    - lookup-by-name
    - lookup-by-ID

The queue data structure was reused from a pre-existing program. The set data structure was implemented very simply as a fixed-size bit vector. The symbol table data structure was implemented as a fixed-size array. This allowed a quick implementation, but provided a very useful interface, and would allow for future improvements in the implementation.

Other generally useful packages were developed (to the extent required for the implementation), such as a set of file and directory name manipulation routines, and data structures to handle mapping of object or configuration names to file names.

The use of abstract data types as building blocks, attention to reusability issues in the design of the parser, lexical analyzer, and configuration data structures, and the sharing of data structures and code between occ and acp were instrumental in producing higher functionality with less effort.

The ability to implement the AOL constructs in a high level language (C) rather than in machine language provided considerable implementation leverage as well. For example, the mapping from object type and operation name to an operation index is implemented by accessing the index using a global variable name formed mainly by the concatenation of the object type name and the operation name. At system configuration time, acp generates a set of initialized declarations of these names with the proper values. This allows separate compilation by making the linker do most of the work. As another example, occ can use the C compiler to do some of the work of type checking by generating code that will cause type errors to be detected by cc if the types specified in AOL constructs were incorrect. cpp is used to implement some AOL constructs. The C compiler also provides the **asm** construct when machine language is required.

5.2 The AOL Preprocessor

This section describes the operation of occ. The function of occ is to read an object definition, doing syntax and type checking, translate it into standard C, and run the C compiler to compile the resulting code. The user runs occ in a manner similar to that in which the C compiler (cc) is run. In addition to occ-specific command line switches (described in [Northcutt 88d], the user should specify the cc-specific switches that should be used to compile the object.

Object type names are mapped to file names using a user-supplied *map* file. This file is given as input to occ, along with the name of the object type whose object definition is to be processed.

occ is implemented in two passes. Once it has located the object definition to be processed, it performs the first pass on that object definition. The first pass stores information concerning the object's name, attributes (e.g., kernel, client, system), operations and their formal parameters, well-known capability names, and the names of object types used in capability declarations. occ then runs the first pass on all the object definitions of the object types accessed by the object definition being translated. This provides the information necessary to check the operation names and type-check the parameters used in **invoke** statements.

The second pass of occ generates and outputs the code for the AOL constructs when they are encountered in the input; the lexical analyzer takes care of passing through the non-AOL C code to the output. The implementations of these constructs in standard C are fairly straightforward, but rather system-specific. Descending to the low level of detail necessary to describe them is inappropriate in this document. Suffice it to say the AOL constructs are translated into in-line C code, which uses library routine calls that perform the system traps that actually constitute the interface to the kernel (which is hidden by AOL). When the generated C code declares named entities, it uses a prefix of "OCC_" to attempt to prevent conflicts with user-declared names.

The second pass also does the type checking of invocation arguments. If type errors are found during translation, occ reports the error (with the appropriate line number) and exits. (Any syntax errors would have been found in during the first pass, and handled similarly.) If the second pass completes without errors, occ runs the C compiler on the resulting code, with the user-supplied switches, to produce the binary version of the object definition. If the C compiler finds errors, it reports them to the user, again with the appropriate line numbers corresponding to the AOL source file.

5.3 The Alpha Configuration Program

The function of acp is to take as input a configuration definition and the object definitions that comprise an Alpha application system and produce the code, data, and command files that will produce load modules for the testbed system nodes. As noted above, acp shares much code with occ, including the AOL parser, configuration representation data types, etc.

We now describe the operation of acp. The user creates a directory suitable for building the system and executes acp from that directory, giving it the name of the configuration to be built. (Details relating to the use of acp and other tools used in building the system are given in [Northcutt 88d]). The acp program copies various system *makefiles* to the current (build) directory, and executes a tool that generates the make dependency rules for the kernel source files and for the application source files.

acp then begins building an internal representation of the configuration. It starts by including in the configuration the system objects provided by Alpha as the interface to system services, and their well-known capabilities. It also includes standard kernel objects that are part of the kernel implementation. After this initialization, acp parses the configuration definition and enters the specified nodes, object instances, initial threads, and well-known capability mappings into the configuration.

A number of object types have been referenced by the configuration definition. These objects must be *defined* by reading their definitions (essentially by using the first pass of occ) to determine relevant information about their operations, well-known capabilities, and references to other objects. These objects in turn will reference other objects, which must be defined as well. This process continues until there are no undefined objects in the configuration.

When all the objects are successfully defined, the configuration is analyzed and checked for consistency as follows. acp assigns internal identifiers to the well-known capabilities for system objects, object types, and to user-defined well-known capabilities. All well-known capability names are resolved to determine their corresponding internal identifier. Then, all of the well-known capability declarations are checked to determine whether the types declared for the well-known capabilities correspond to the type of the object finally mapped to the well-known capability name. (The type information for the well-known capability declarations is part of the configuration information for each object, determined when the objects were defined.)

Also, acp checks that, for any node, only one instance of a particular kernel object type is specified. It checks that the object type specified for each instance defined in the configuration has been defined. It also checks that the initial operation specified for each thread defined in the configuration is a valid operation, and that the types of the initial invocation parameters are correct.

At this point, acp is ready to configure the system; this activity consists of several steps. First, it generates C source files and data files containing data structures that describe all of the object types to the kernel, so that it can create instances of them and supply those instances with their well-known capabilities. It generates a C source file that contains initialized declarations of global variables that provide the mapping from (object-name, operation-name) pairs and (well-known-capability-name, operation-name) pairs to operation indices used in invocations.

acp then generates an initialization object for each node. After the kernel has initialized itself, it will create a thread that invokes the initialization object. This object creates the object instances and initial threads specified in the configuration definition, initializes the well-known capability data structures in the kernel (by invoking operations on a special system object) and starts the initial threads running.

Finally, acp generates a makefile containing the rules that will compile the system and application, link the kernel and object types, and create load modules for each node in the system.

At this point the user has only to type “make” and the system will be built. Then the user can boot and run the system, as described in [Northcutt 88d].

6 Acknowledgments

Duane Northcutt did an early design and implementation of a subset of this facility, which established some precedents carried forward in the current work.

Sam Shipman led the redesign effort, with substantial input from Duane Northcutt and Ray Clark. Sam Shipman implemented `occ`, `acp`, and other tools. Ray Clark implemented kernel support for well-known capabilities. Duane Northcutt, Ray Clark, and Huay-Yong Wang implemented other tools that made the complete automation of the user development cycle possible.

David Maynard provided valuable input, and relayed feedback from users at General Dynamics, which resulted in improvements in the design and implementation.

References

[Accetta 86] Accetta, M. J., Baron, R. X., Golub, D., Rashid, R. F., Tevanian, A. and Young, M.
A New Kernel Foundation for UNIX Development.
In *Proceedings of the Summer 1986 USENIX Technical Conference and Exhibition*, pages 35-41, June, 1986.

[Clark 88] Clark, R. K., Kegley, R. B., Keleher, P. J., Maynard, D. P., Northcutt, J. D., Shipman, S. E. and Zimmerman, B. A.
An Example Real-Time Command and Control Application.
Archons Project Technical Report #88032, Department of Computer Science, Carnegie-Mellon University, March, 1988.

[Jensen 75] Jensen, E. D.
Time-Value Functions for BMD Radar Scheduling.
Technical Report, Honeywell System and Research Center, June, 1975.

[Jensen 88] Jensen, E. D.
Alpha: A Real-Time Decentralized Operating System for Mission-Oriented System Integration and Operation.
In *Proceedings, RICIS Symposium '88, Integrated Computing Environments for Large, Complex Systems*, NASA, November, 1988.

[Locke 86] Locke, C. D.
Best-Effort Decision Making for Real-Time Scheduling.
Ph.D. Thesis, Department of Computer Science, Carnegie-Mellon University, May, 1986.

[Northcutt 87] Northcutt, J. D.
Mechanisms for Reliable Distributed Real-Time Operating Systems: The Alpha Kernel.
Academic Press, Boston, 1987.

[Northcutt 88a] Northcutt, J. D.
The Alpha Operating System: Requirements and Rationale.
Archons Project Technical Report #88011, Department of Computer Science, Carnegie-Mellon University, January, 1988.

[Northcutt 88b] Northcutt, J. D. and Clark, R. K.
The Alpha Operating System: Programming Model
Archons Project Technical Report #88021, Department of Computer Science, Carnegie-Mellon University, February, 1988.

[Northcutt 88c] Northcutt, J. D.
The Alpha Distributed Computer System Testbed.
Archons Project Technical Report #88033, Department of Computer Science, Carnegie-Mellon University, March, 1988.

[Northcutt 88d] Northcutt, J. D. and Shipman, S. E.
The Alpha Operating System: Programming Utilities.

Archons Project Technical Report #88041, Department of Computer Science, Carnegie-Mellon University, April, 1988.

[Northcutt 88e] Northcutt, J. D.
The Alpha Operating System: Kernel Programmer's Interface Manual.
Archons Project Technical Report #88061, Department of Computer Science, Carnegie-Mellon University, June, 1988.

Appendix I: AOL Grammar

The notation used to express the AOL syntax consists of terminals, non-terminals, and metacharacters. Terminals are rendered in a boldface Helvetica font (e.g., **INVOKE**), non-terminals in italic Times Roman (e.g., *object_definition*). The metacharacters are larger than the others, and consist of parentheses, brackets, and braces. Alternatives are vertically stacked. Constructs enclosed in brackets [] are optional. Braces { } denote zero or more repetitions of the enclosed construct (a Kleene star operator). Parentheses are only for grouping alternatives. Also, non-terminals ending in "*_name*" denote identifiers (i.e., a letter followed by zero or more letters or digits).

Thus, in the object definition example, the **CLIENT**, **KERNEL**, and **SYSTEM** modifiers are mutually exclusive, and optional. If they were enclosed in parentheses, one and only one of them would be required. Any sequence of the non-terminals enclosed in large braces is legal (including the empty one). Note that, in the syntax diagrams, we make no effort to unify AOL syntax with that of the C language—we are only concerned with the AOL extensions in the syntax descriptions. The relationships between AOL constructs and C are made clear in the text of this report.

AOL Grammar

Top-level Constructs

configuration_definition
object_definition

aol_language

Configuration Definition

CONFIGURATION *configuration_name* **{** *use_statement* / *map_statement* / *node_definition* **}**

configuration_definition

USE *object_type_name* **,** *object_type_name* **;**

use_statement

MAP *capability_name* **TO** *instance_name* **;**

map_statement

AOL Grammar (continued)

$$\textbf{NODE}\ \textit{node\_name}\ \textbf{\{}\ \left[\left\{ \begin{array}{c} \textit{use\_statement} \\ \textit{instance\_definition} \end{array} \right\} \right]\ \textbf{\}}$$

node_statement

$$\textbf{INSTANCE}\ \textit{instance\_name}\ \textbf{OF}\ \textit{object\_type\_name}\ \textbf{\{}\ \left[\left\{ \textit{thread\_statement} \right\} \right]\ \textbf{\}}$$

instance_definition

THREAD *thread_name* **INVOKES** *operation_name* **(** *actual_parameter_list* **) ;**

thread_statement

Object Definition

$$\left[\begin{array}{c} \textbf{CLIENT} \\ \textbf{KERNEL} \\ \textbf{SYSTEM} \end{array} \right]\ \textbf{OBJECT}\ \textit{object\_name}\ \textbf{\{}\ \left[\left\{ \begin{array}{c} \textit{use\_statement} \\ \textit{capability\_declaration} \\ \textit{operation\_definition} \end{array} \right\} \right]\ \textbf{\}}$$

object_definition

$$\left[\textbf{WELLKNOWN} \right]\ \textbf{CAPA}\ \left[\textit{object\_type\_name} \right]\ \textit{capability\_name}\ \textbf{;}$$

capability_declaration

AOL Grammar (continued)

Operation Definition

$$\textbf{OPERATION}\ \textit{operation\_name}\ \textbf{(}\ \textit{formal\_parameter\_list}\ \textbf{) \{}\ \left[\left\{\begin{array}{c} \textit{use\_statement} \\ \textit{capability\_declaration} \\ \textit{invoke\_statement} \\ \textit{return\_statement} \\ \textit{deadline\_block} \\ \textit{exception\_block} \end{array}\right\}\right]\ \textbf{\}}$$

operation_definition

$$\textit{formal\_parameter}\ \left[\left\{\ \textbf{,}\ \textit{formal\_parameter}\ \right\}\right]$$

formal_parameter_list

$$\left(\begin{array}{c} \textbf{IN} \\ \textbf{OUT} \\ \textbf{IN OUT} \end{array}\right)\left(\begin{array}{c} \textit{C\_type\_specification} \\ \textbf{CAPA}\ \left[\textit{object\_type\_name}\right] \end{array}\right)\ \textit{name}$$

formal_parameter

AOL Grammar (continued)

Invoke and Return Statements

[**UNCHECKED**] **INVOKE** *capability_name* **.** *operation_name* **(** *actual_parameter_list* **)** { **;** | *failure_clause* }

invoke_statement

actual_parameter [{ **,** *actual_parameter* }]

actual_parameter_list

[**IN** | **OUT** | **IN OUT**] [**(** { *C_type_specification* | **CAPA** [*object_type_name*] } **)**] *C_expression*

actual_parameter

ON FAILURE { { *use_statement* | *capability_declaration* | *invoke_statement* | *return_statement* | *deadline_block* | *exception_block* } **}**

failure_clause

AOL Grammar (continued)

$$\textbf{RETURN} \left\{ \begin{array}{c} \textbf{SUCCESS} \\ \textbf{FAILURE} \end{array} \right\} \left[\textbf{(} \; \textit{C\_expression} \; \textbf{)} \right] \textbf{;}$$

return_statement

Deadline and Exception Blocks

$$\textbf{DEADLINE (} \textit{deadline\_arguments} \textbf{) \{} \left\{ \begin{array}{c} \textit{use\_statement} \\ \textit{capability\_declaration} \\ \textit{invoke\_statement} \\ \textit{return\_statement} \\ \textit{deadline\_block} \\ \textit{exception\_block} \end{array} \right\} \left[\textit{exception\_clause} \right]$$

deadline_block

$$\textbf{BLOCK \{} \left\{ \begin{array}{c} \textit{use\_statement} \\ \textit{capability\_declaration} \\ \textit{invoke\_statement} \\ \textit{return\_statement} \\ \textit{deadline\_block} \\ \textit{exception\_block} \end{array} \right\} \left[\textit{exception\_clause} \right]$$

exception_block

$$\textbf{ON EXCEPTION \{} \left\{ \begin{array}{c} \textit{use\_statement} \\ \textit{capability\_declaration} \\ \textit{invoke\_statement} \\ \textit{return\_statement} \end{array} \right\} \textbf{\}}$$

exception_clause