An Overview of DOD-STD-1838A (proposed), The Common APSE Interface Set, Revision A

Robert Munck
The MITRE Corporation

Patricia Oberndorf
Naval Ocean Systems Center

Erhard Ploedereder, Ph.D.
Tartan Laboratories

Richard Thall
SofTech

Abstract

A five-year effort under the Ada Joint Program Office has developed a proposed standard for a host system interface as seen by tools running in an Ada Programming Support Environment (APSE). Standardization of this interface as DOD-STD-1838A will have a number of desirable effects for the Department of Defense, including tool portability, tool integration, data transportability, encouragement of a market in portable tools, and better programmer productivity.

As the capability of tools to communicate with each other is a central requirement in APSEs, the Common APSE Interface Set (CAIS) has paid particular attention to facilitate such communication in a host-independent fashion. CAIS incorporates a well-integrated set of concepts tuned to the needs of writers and users of integrated tool sets.

This paper covers several of these concepts:

- the entity management system used in place of a traditional filing system,
- object typing with inheritance,
- process control including atomic transactions,
- access control and security,
- input/output methods,
- support for distributed resource control, and
- facilities for inter-system data transport.

Background

Early in the development of the Ada language, it was recognized that a computer-based programming support environment was a practical necessity for Ada programmers. It is now clear that many of the promised advantages of Ada, including reusability of code and ease of maintenance, would require a level of commonality in the *programming environment* similar to that provided by the language. This results from the fact that the product of a programming project is not a single executable-form module, but a large, inter-related set of source files, design documents, test plans, and many other kinds of information that must also be present as part of the final product; some of these sets will be drawn from libraries of reusable components. Moreover, the *tools* that the original programmers use to store and manipulate these files and relationships among files must also be available to their successors who wish to reuse or maintain the product.

The concept of an Ada Programming Support Environment was well developed in the "Stoneman" document [STONEMAN80]. It specifies an architecture containing an identifiable interface between code having local host or operating system dependencies and code that would be portable from one host to another. The code implementing this interface is called the Kernel APSE or KAPSE. The following text from the **CAIS Reader's Guide** [CAIS87] describes the DoD effort to develop a standard for that interface:

"When DoD started procuring tools for the Ada program, it did not restrict itself to procuring individual tools. Rather, the DoD embarked upon the procurement of APSEs. Two procurements were started: one by the Army, called the Ada Language System (ALS), and the other by the Air Force, called the Ada Integrated Environment (AIE). Unfortunately, the interfaces provided (by) the KAPSE . . . were different in these two APSEs. Because of divergent approaches at the KAPSE interface level by the ALS and AIE contractors, a team was formed . . . to define more specific KAPSE interface requirements. This team is the KAPSE Interface Team (KIT) and is chaired by the Naval Ocean Systems Center (NOSC), a Navy laboratory. Added to the KIT was the KAPSE Interface Team from Industry and Academia (KITIA). The KIT/KITIA (produced) DOD-STD-1838, the Military Standard Common Ada Programming Support Environment (APSE) Interface Set." [CAIS86]

In parallel with the development of the prototype standard DOD-STD-1838, the KIT/KITIA developed the Requirements and Design Criteria document (RAC) [RAC86] and a Rationale discussing its contents [RAT87]. A contractor was then competitively chosen to evolve the CAIS as indicated by the RAC, experience with the original CAIS and other APSEs, and advances in the state of the art of Software Development Environments (SDEs) since the work was begun. That revision, sometimes called CAIS-A, is in the formal process needed to update DOD-STD-1838 to DOD-STD-1838A; after the review procedure, it will become the official standard, probably sometime in 1989. The term CAIS is used throughout this document to refer to that proposed revision of the standard.

As with Ada, the CAIS must be backed by a thorough test for adherence of an implementation to the standard. The mechanics of validation (an evolving validation suite, validation centers, derived validations on similar hosts, etc.) will be much the same as Ada compiler validation. As with the ACVC, CAIS validation will show compliance with the specification, but will not reveal the usability (response, capacity, etc.) of the implementation.

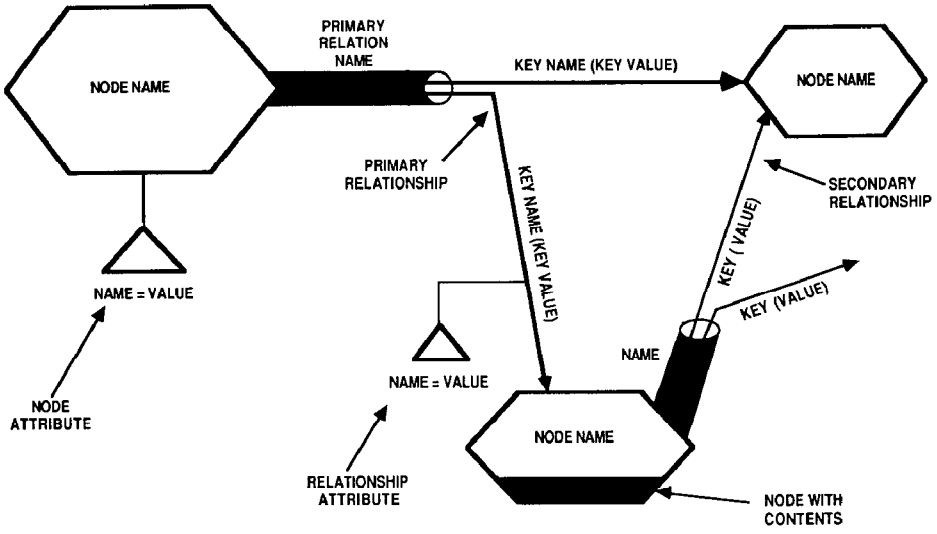
Introduction

The CAIS design represents well in excess of 100 man-years of work by experts in software development environments, operating system, data base systems, and programming languages from industry, academia, and government in the US and Europe. These experts are generally the most senior technical people from their organizations, active in research and development of SDEs. The expertise brought to bear was extraordinary; the arguments were epic.

The technical and institutional requirements that the CAIS attempts to meet (fairly successfully) are quite wide-ranging; the resulting design is large and complex. An attempt to summarize it within the constraints of a technical paper made necessary a number of short-cuts. The paper presents a cursory overview of the major features of the CAIS. A more detailed rationale of the design choices and a discussion of their interactions is beyond the scope of this paper. The reader is referred to the Rationales for DOD-STD-1838 [RAT88] and the RAC for many of the basic decisions of the CAIS model.

The interrelation of the features of the CAIS, its *integration*, was one of the driving topics of the majority of the design work. Long, hard discussions took place on interaction of the process model with transactions, of strong data typing with the naming mechanism, and so forth. The result, summarized below, is in fact very tightly integrated; conflicts between features have been ruthlessly searched for, discussed, and solved, often by clever and insightful design ideas.

The Basic Structure



The CAIS is based on an Entity-Relationship-Attribute (ERA) model in which *nodes* (entities) are connected by *relationships* (edges) forming a general graph. Both nodes and relationships may have *attributes* which are name-value pairs. In this and the following figures, nodes are represented by trapezoids, relationships by arrows between them, and attributes by triangles attached to the node or relationship.

When we refer to "names" in this description of the CAIS, we mean a concept that allows the user the identification of some entity. Such "names" need not be identifiers in the form of strings, and they generally are not in the CAIS. We shall return to the naming issues in a later section.

Relationships may be grouped together under a single name; the group is called a *relation*. All relationships in a relation are of the same *type* (discussed later), all have the same set of attributes but with possibly different values.

A particular variety of relationship attributes, called *keys*, is used to disambiguate among the many relationships of a relation that may emanate from a node. The relationships of a relation may have more than one key attribute. For selecting a relationship emanating from a

node, the relation and a set of disambiguating key attributes and their value ranges are specified. The value of a key attribute is a string with the syntax of an Ada identifier. Since each key attribute is identified as part of the selection, their values need not be disjoint for different attributes.

Relationships may be one-way, pointing from a source node to a target node, or two-way. Two-way relationships are effectively two one-way relationships pointing both ways, except that they cannot be created or deleted separately; both must exist if either does.

Relationships between nodes are the primary means of identifying nodes. Once a relationship emanating from a given node is selected, the target node of that relationship is uniquely identified. The process of "walking the graph" to identify nodes is called "navigation". Once a node is identified, a handle can be obtained which will continue to refer to the node until explicitly released.

Nodes and relationships may form a general graph or "bowl of spaghetti." However, this raises various practical problems of deletion and garbage collection, long-term naming, and unconnected sub-graphs. CAIS therefore designates certain relationships as *primary* (and all others as *secondary*) and requires that all nodes and primary relationships in the data base form a single tree structure. This means that every node other than the root is pointed to by one and only one primary relationship, allowing the further definition that a node is deleted when the primary relationship pointing to it is deleted. Secondary relationships pointing to a deleted node become unusable for accessing the target node.

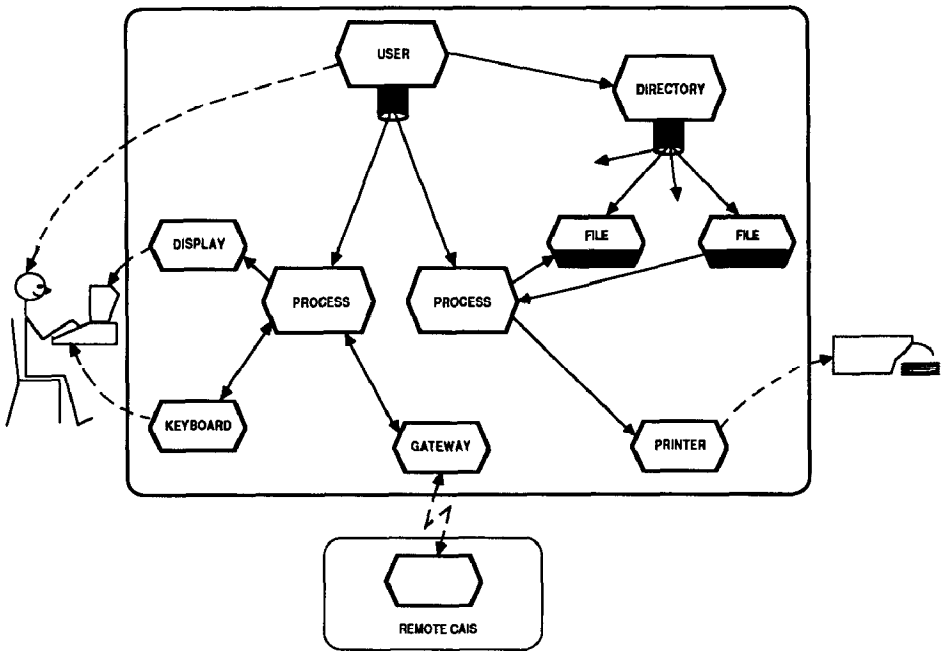
Primary relationships are required to be two-way. A node may have many outgoing primary relationships in different relations. The effect is that a "directory" node can have several sets of nodes identified by different primary relations, not just one.

Attribute values may be INTEGER, FLOAT, STRING, IDENTIFIER, and composites of those types. Composites may be all of a single primitive type, similar to an array, or mixed types, similar to an Ada record. As with Ada, variant components within a record are possible.

Nodes may have "contents," data of unknown format (to the CAIS) that can be read and written by use of CAIS I/O facilities.

It is intended that nodes be used to represent "things," relationships represent the associations between those things, and attributes the additional qualities of nodes and relationships. This intent is obviously not enforceable in any way by the CAIS, and it is anticipated that different people will choose different models for similar applications. The structure is sufficiently flexible that many alternative representations may be tried until a "best" one is found.

Ubiquity of the Node Model



The CAIS node model is unique in that **everything** in the computer system of concern to tools is represented in terms of the node model. Not only stored data and structure (files and directories), but users, processes, I/O devices, processors, data paths, network connections, and type definitions are all nodes in the data structure, with appropriate attributes and relationships to other nodes.

There is therefore a single name space in the CAIS, if the idea of a "name" is stretched to include pathnames (sequences of relation names, key names, and key values

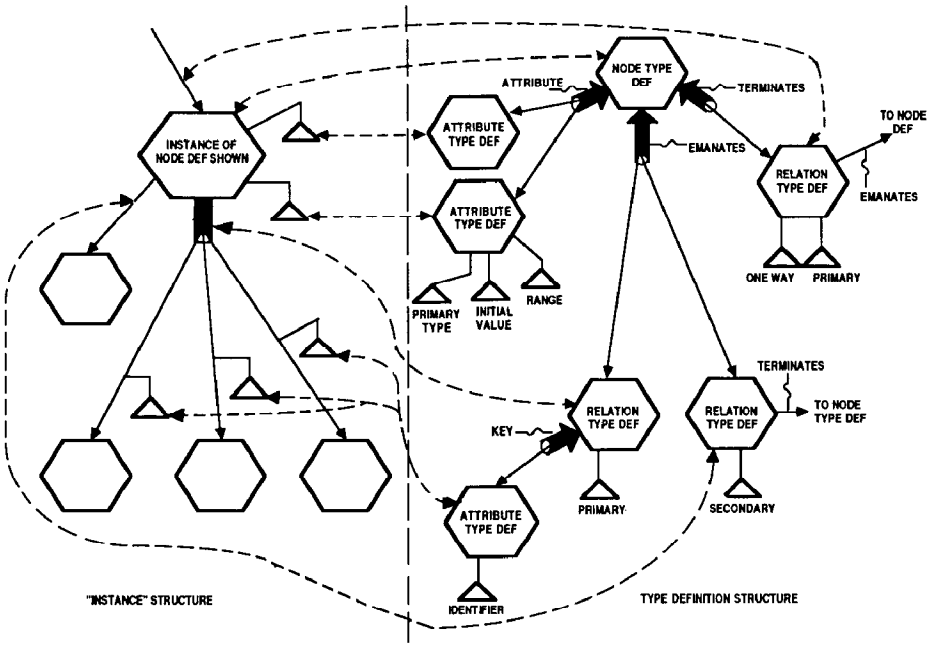
that trace a path through the structure). However, each node is generally reachable via many pathnames. Access control considerations make identification by such pathnames a relatively expensive operation. For this and other reasons of efficiency and functionality, the primary paradigm of identification in CAIS uses handles into the graph structure which were obtained by graph navigation. Attribute "names" are in reality handles to the attribute definitions, relation "names" are handles to the relation definitions, and so forth. This is shown in detail below.

Interestingly, though, the need for character-string names and pathnames becomes less important in a general structure of this nature. It is obviously bad practice to store a name or pathname in the data structure (i.e., in the contents of a node or as the value of an attribute); a relationship is much preferred for the great majority of circumstances. Also, programmers are more likely to be using graphical displays that show a representation of the data structure, in which they can "name" entities by pointing with the mouse.

The figure shows a user with two processes running, one communicating with him through a keyboard and display and with a remote CAIS through a network gateway, and the other manipulating two of his files and printing. As suggested above, the part of the figure inside the large box could actually be displayed in a window on his screen by a command handler tool. Of course, more meaningful icons for the various kinds of nodes could be devised.

(Note: the figure is simplified somewhat in its depiction of I/O connections; see the description of "Channel nodes" below.)

Type Definitions for Data Structure Components



The proposed update of DOD-STD-1838 to -1838A adds the concept of *typing*. Individual nodes, relations, and attributes are said to be instances of a *type* and must conform to a *type definition*. The type definition of a node states what attributes and relations it has and whether or not it has Contents. The type definition of a relation states what nodes it may originate from and point to, what attributes and keys it has, whether it is one- or two-way, how many relationships it may contain, and other constraints. The type definition of an attribute states what primitive types it is made up of, what values it may assume, default or initial values, and whether or not it may be changed by a user.

As with everything else, type definitions are represented by nodes in the data structure. It may be a bit confusing at first that relation and attribute type definitions are represented by **nodes**, but a little thought will show why this is desirable.

The figure shows a piece of "instance" structure and the corresponding "type definition" structure of the large node and its attributes and relationships. As shown, nodes in the example that are instances of that type have two attributes; their definitions are pointed to by the Attribute relation. They have one incoming relationship (at the top), the definition of which is found via the Terminates relation. They have two outgoing relations, definitions of which are found through the Emanates relation from the node type definition node. Finally, one of the outgoing relations has one key attribute.

Note that most of the relations in the type definition structure are two-way; for example, relation type definitions have EMANATES_FROM and TERMINATES_AT relations to the node types that they may connect. These are the "other direction" of the Emanates and Terminates relations on the node type definitions.

The figure is misleading in that it distinguishes between "instance" and "type definition" structure; in fact, both are part of the same structure. Moreover, the relation-

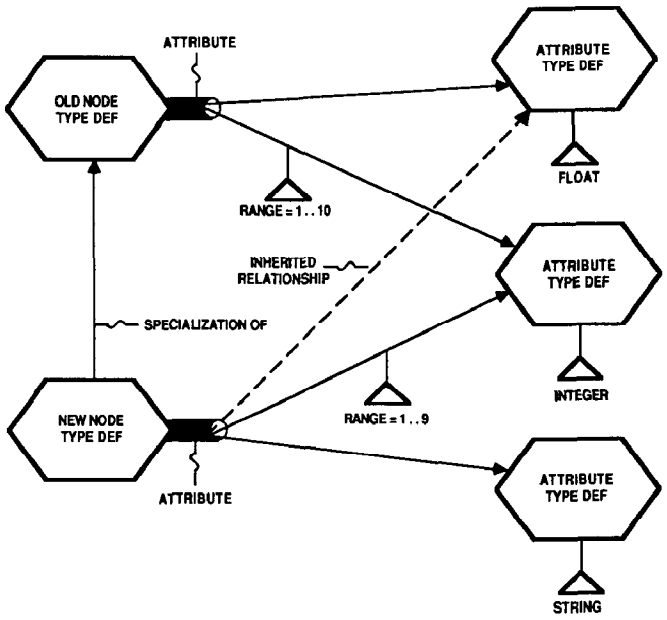
ships, attributes, and nodes in the definition structure are, of course, instances of some type definitions not shown in the figure. For example, there must be a definition node that defines node type definitions, which is an instance of itself and describes the type of all node type definition nodes.

Most tools meant to run on the CAIS will be written to operate on particular types of data base items. They will, in effect, say to the CAIS "I want to access *this* node and expect it to be of *that* type." The CAIS will allow the access if the requested type and the actual type of the node are the same or compatible (discussed below) and signal an error if not. Many user errors that would have undesirable results on conventional untyped systems will be prevented in this way.

SDE data bases normally have a complex set of rules governing their manipulation. These rules are in the form of canned procedures, pre-defined structure and access rights, naming conventions, and programmer directives. Most of them can be broken or bypassed by user error or intent. CAIS allows most of these rules to be expressed explicitly as type definitions and made as unbreakable as management desires for project data or as forgiving as the individual programmer desires for private data.

A more complete discussion of the need for and advantages of typing can be found in [MUNCK88].

Type Specialization



The major difference between the concept of "type" in programming languages and its use in the CAIS is that (typed) objects in the CAIS "last forever" or are *persistent*; they survive between executions of the programs that create and manipulate them. The resulting problem is that definitions need to change in the course of a project. Circumstances change, people make mistakes and correct them, the project moves from phase to phase, and people learn how to do things better. It is usually infeasible for all tools and data using a single type definition to change at one time, especially if the tools and data are spread over many installations.

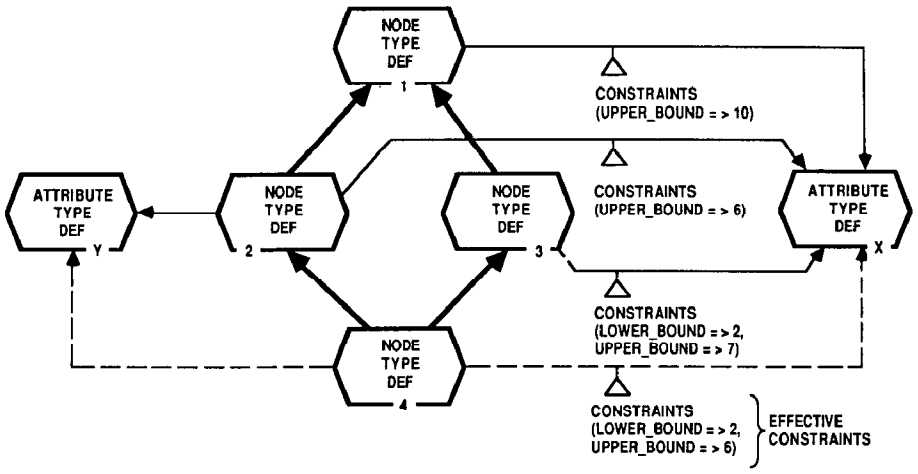
If a definition is changed while instances of it exist, there may be an inconsistency between the (new) definition and the (old) objects. One solution to this problem, used in the CAIS, is that definitions are not generally changed, but rather new versions of them are created and co-exist with the old. If the changes that can be made to a type definition to produce a new version are properly restricted, tools compiled to access objects of the old type will work correctly on instances of the new type definition version. In the CAIS, such a new version is called a *specialization* of the old. A Specialization_Of relationship connects the new definition to the old one.

The figure shows a simple example of Specialization, in which the old node type definition specified a FLOAT attribute and an INTEGER attribute with a range of 1..10. The new version restricts the INTEGER attribute's range to 1..9 and adds a STRING attribute. Note that the new version *implicitly* inherits the FLOAT attribute; the relationship indicated by the dotted line does not actually exist.

Tools coded to operate on instances of the old definition will continue to work on instances of the new; they are unaware of the additional STRING attribute. However, an attempt to set the INTEGER atribute to 10 will cause an exception. This exception is probably what the person who created the new definition wants to have happen. Some circumstance caused him to reduce the range, and that same circumstance almost certainly makes the exception desirable. Old tools that only read the INTEGER attribute and new tools that do not reference the STRING attribute will work on instances of both the old and new definitions.

The inverse relation to Specialization_Of is the relation Generalization_Of. Types can be created as generalization of already existing more specialized type definitions. Thus it becomes possible to develop tools operating on common properties of objects whose types hitherto were unrelated.

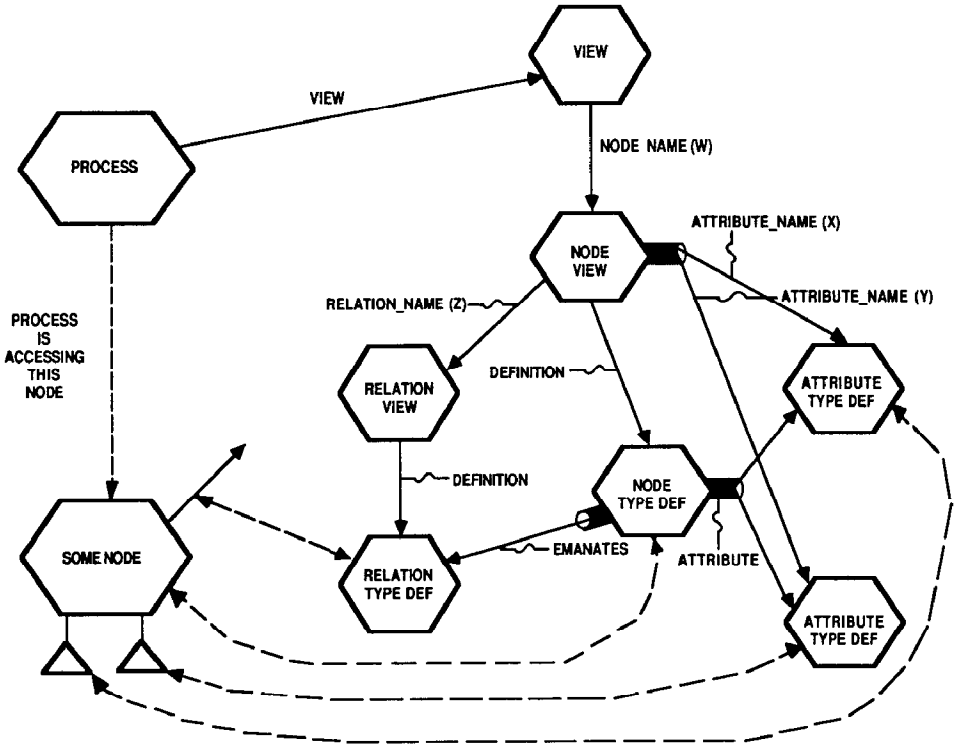
Specialization from More than One Definition



Specialization adds components to or tightens the constraints on a definition. It is therefore possible to form the specialization of two or more definitions if their components add together without conflict and a "tightest common constraint" can be found for all constraints.

The figure shows Node Definition 4 as the specialization of Definitions 2 and 3. The attribute added by Definition 2 (Y) is inherited by 4 and the constraints on 4's attribute (X) are the "sum" of those defined by 1, 2, and 3.

The Naming Mechanism



Tools running on an SDE need to name the items in the data base that they want to access. The host system must have some way of associating names with the items so that these names can be embedded in tool code that uses them. Obviously, this can be a source of problems when moving a tool from one system to another, as the naming mechanisms or conventions of the two systems may be different.

UNIX has managed to achieve a fair portability of tools, but the ways the naming problem is usually solved when moving a tool from one UNIX to another do not meet CAIS requirements. Most porting of small, self-contained UNIX tools is achieved by providing source code of the tool to the new system. The embedded names are manually changed as appropriate and the tool (re-)compiled.

This approach makes it difficult or impossible for vendors to create and sell tools profitably, because distribution of source makes pirating of the software or of its design too easy. An active market in tools cannot exist if source code must be distributed, and indeed no significant commercial market in small tools exists for UNIX. Such a market for CAIS is a very important goal of the DoD.

Note that there is an active market in tools for MS-DOS and Apple systems. Both of these are so small and simple that very few names need to be used, and tools can be written to accept them as parameters. A market also exists in large tools or tool sets for UNIX, in which the tools are sufficiently self-contained that most of the names they use are defined by the tool. The relatively few local names that they need are specified during an installation process, passed as the values of environment variables, or passed as parameters.

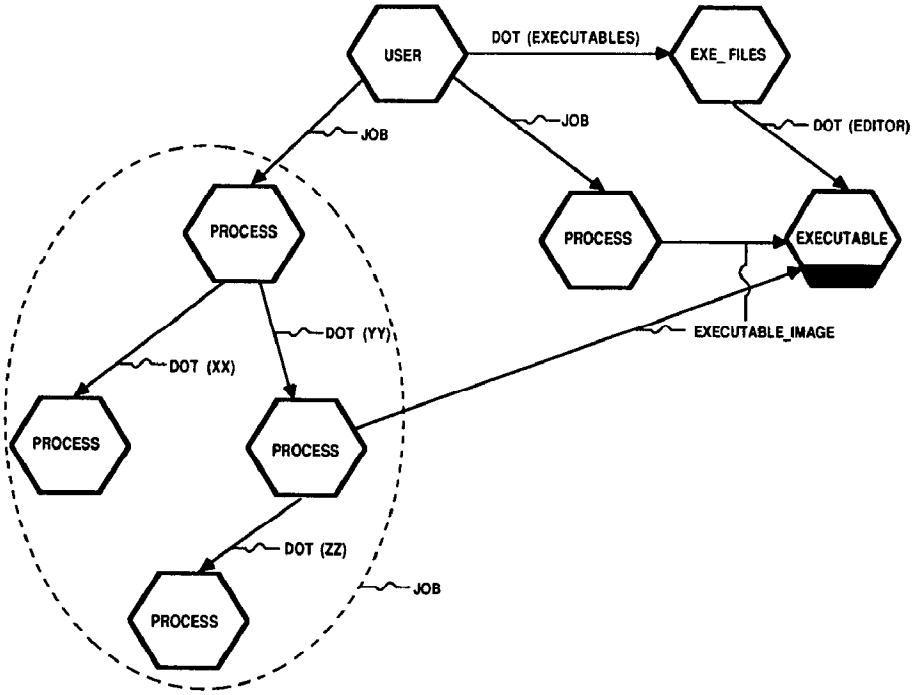
It is therefore necessary that the CAIS have a naming mechanism that does not necessitate re-compilation of tools when name changes are necessary and that allows each tool to have private, non-standard names for items in the data base. This is accomplished by the data structuring facilities and as shown in the figure with what are called *views* or *interpretations*.

The view structure is essentially a parallel copy of the type definition structure with the type definition data stripped off. Each node in the view structure has a DEFINITION relationship to the type structure node to which it corresponds. The figure shows a "Node View" and a "Relation View" (center) and the corresponding "Node Type Definition" and "Relation Type Definition." There is no need for "Attribute View Nodes;" Node View and Relation View nodes have relationships directly to the corresponding Attribute Type Definition.

The names used by a tool are specified as key values on the relationships in the view structure. For example, the process in the figure (upper left) can refer to the two attributes on the node (lower left) as "X" and "Y" and to the relation as "Z". These are the values of the Attribute_Name and Relation_Name keys on the relations from the Node view to the Relation View and two Attribute Type Definitions.

The CAIS allows multiple names for items by allowing multiple views covering any particular section of the type definition structure. Views can omit items, giving tools no way to name them; for example, a view can make a relation "invisible" by not having a DEFINITION relationship to the Relation Type Definition. Because the ability of tools to use particular views can be restricted through the access rights mechanism (see below), the creators of the view structure have a great deal of control over data structure access.

Process Nodes and Jobs



A CAIS *Process* is approximately a running Ada Main Program ("approximately" because there is no requirement that it be written in Ada). Processes have one or more internal threads of control, i.e., Ada tasks. For the CAIS to be a portability platform, it is impossible for it to say very much about the internal functioning of a process; compiler and operating system vendors cannot be overly restricted. The strongest statement made, and a very controversial one, is that other tasks in a process must continue to run when one is blocked awaiting fulfillment of a CAIS request. For example, having one task wait for the next keyboard value entered must not stop the other tasks until a key is hit.

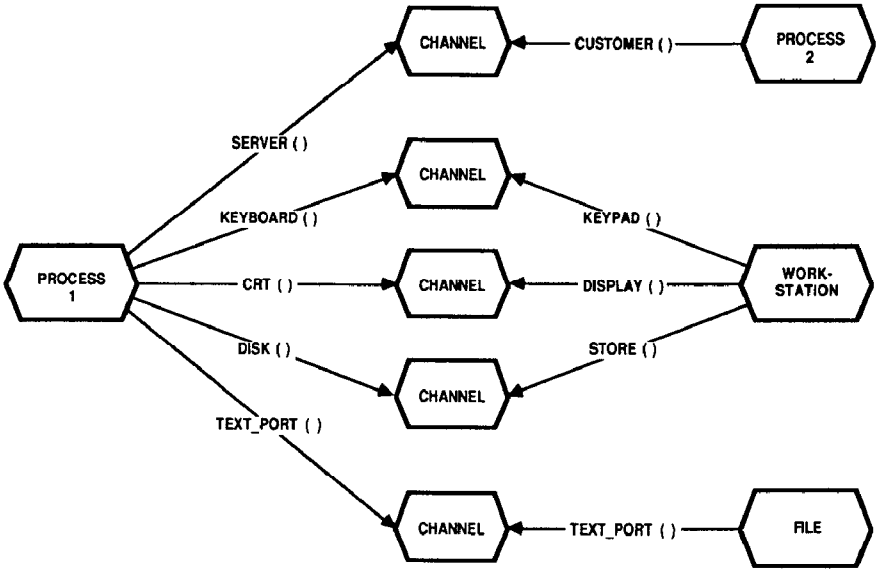
Processes may request that other (dependent) processes be started; they thus form a hierarchical tree of parents and children. Such trees are called "Jobs" and are rooted at the User Node of the person for whom the processes are running. Similarly, processes may start independent processes. The figure shows a four-process job and a one-process job.

Processes may be suspended, resumed and aborted by request of other processes within access control constraints. Termination of a process will wait for dependent processes to terminate. The process nodes remain after termination of the process and allow recordings of the execution results and statistics. They can subsequently be deleted explicitly.

A process represents the execution of a code file. That file is also a node represented in the node model. The process node is connected to the code file by an Executable_Image relationship. The figure shows two processes running one code file and omits the other process's code.

A mechanism called *Attribute Monitors* will "watch" one or more attributes anywhere in the data structure and trigger execution of a user process when the value of one of them is changed.

Channel Nodes



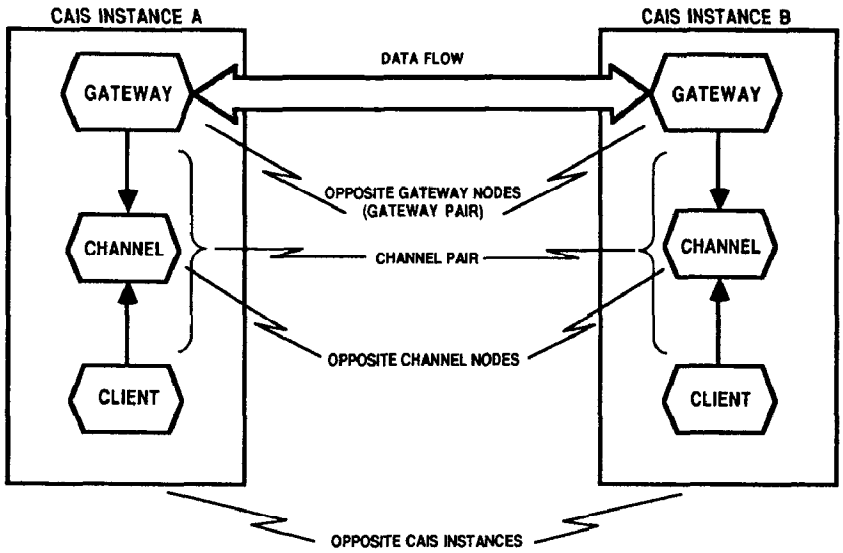
CAIS processes may perform I/O to files, devices, gateways (discussed below), and other processes; it is not generally necessary for a process to know which of these is "on the other end." *Channel* nodes represent the I/O connection explicitly in the data structure.

The figure shows a process (left) doing I/O to another process, a work station, and a file. The work station is accepting or producing three I/O streams, from a keyboard, to a display and from/to a disk. These three streams could alternatively go to three separate device nodes. At least one of the nodes connected to a Channel must be a process; direct device-to-device or file-to-file I/O (and the other combinations) are not supported.

The relationships from processes, files, devices, and gateways have attributes (not shown) that specify how the I/O operations are to be effected. Channels can be used to create arbitrarily-complex data paths among processes, or "data-flow diagrams," in the data structure, a powerful generalization of UNIX's pipes. The communications structure among tools may be set up by other tools (third parties).

The representation of channels may be used in conjunction with the type definition capability to differentiate the semantic content of data streams at various levels. For example, type definitions could be created that allow a print tool to read and print any text file, but restrict an Ada compiler to reading those text files which contain Ada source code.

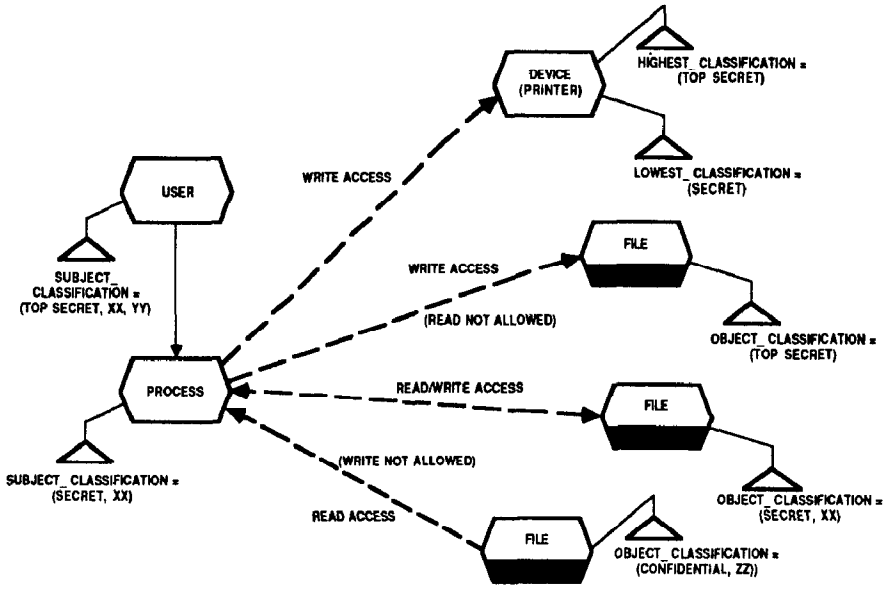
Remote Interprocess Communication



As the figure shows, I/O between processes in different CAIS's is accomplished by *gateway* nodes in each CAIS. The internal workings of gateways and the manner in which they are connected to each other is left to the CAIS implementor. Gateway nodes are passive nodes to which only process nodes can establish channels. This model ensures that cross-CAIS communication can authenticate access rights for data requests across gateways.

Note that gateways bridge separate CAIS implementations, not separate processors running a single distributed CAIS. A CAIS implementation is defined as an instance of the node model with a single root. It may be resident on several processors, in which case the CAIS implementation bears responsibility for the distribution made visible to the user and tools by means of resource maps represented in the node model.

Mandatory Security



One of the more difficult DoD requirements of the CAIS was that it be implementable on (or "as") a Trusted Computer Base (TCB) as a Multi-Level Secure (MLS) system that meets the B3 certification criteria as defined in DoD-5200.28.STD. In more common terms, it must be possible to assign security levels like "Confidential" and "Top Secret" to data in the system and prevent people from accessing data that they are not cleared to access. Multi-level mandatory security is optional in CAIS implementations; they can pass validation without it.

A secure CAIS must support security *categories* in addition to security levels. Categories are used for a number of things, of which a good example is "need to know" access control. A particular file may be Top Secret on a "need to know" basis; this means that users with Top

Secret clearances cannot access it unless they also have been given "need to know" for that particular data or class of data. The security level of the data would be (Top Secret, XYZ_ntk) (where "XYZ_ntk" is an arbitrary label that means "need to know this class") and the user would have to have "XYZ_ntk" in his clearance.

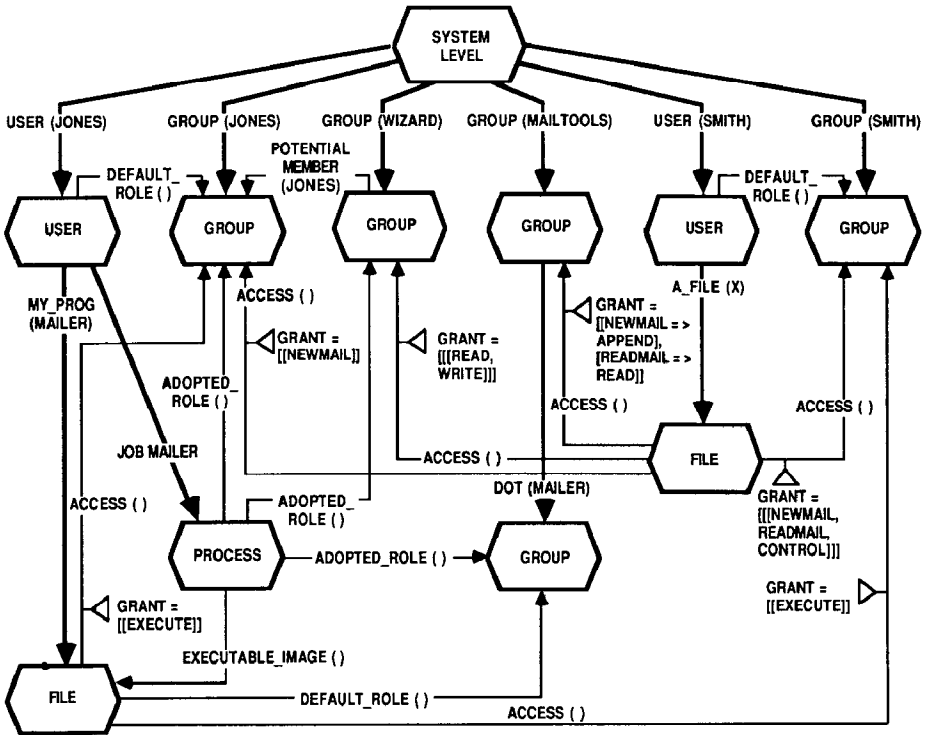
As the figure shows, User nodes have an attribute Subject_Classification containing the user's clearance level and categories. These can be set only by a designated Security Officer and should be the actual clearance level of the user (or lower).

When a user starts a process, that process is said to be an agent of the user, and so it also has a Subject_Classification. The user can choose to give the process a lower clearance, meaning a lower clearance level or a subset of his categories, but not a higher one. The figure shows a User with (Top Secret,XX,YY) who has started a process with (Secret,XX).

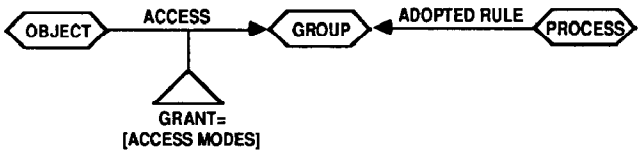
The rules of mandatory security say that a process can read data at security levels less than or equal to the process's and can write data at security levels greater than or equal to the process's. The effect is as shown; the process can read but not write the Confidential file, can read and write the Secret file (and optionally assign it category XX), and can write but not read the Top Secret file. In practical terms, the latter rule means the process can create, over-write, or append data to the file.

Devices are said to have a range of security levels that they can handle. The range is determined by the physical characteristics, location, and I/O path of the device; the printer shown in the figure is probably in a locked and guarded room and, if the processor is elsewhere, connected to the processor by an armored and encrypted cable. The process shown can write Secret or Top Secret data to the printer.

Discretionary Access



The CAIS discretionary access mechanism is based on familiar constructs, expressed explicitly in the node model. Its basic concept can be summarized like this:



The process on the right is allowed to access the object on the left in the ways specified in the GRANT attribute.

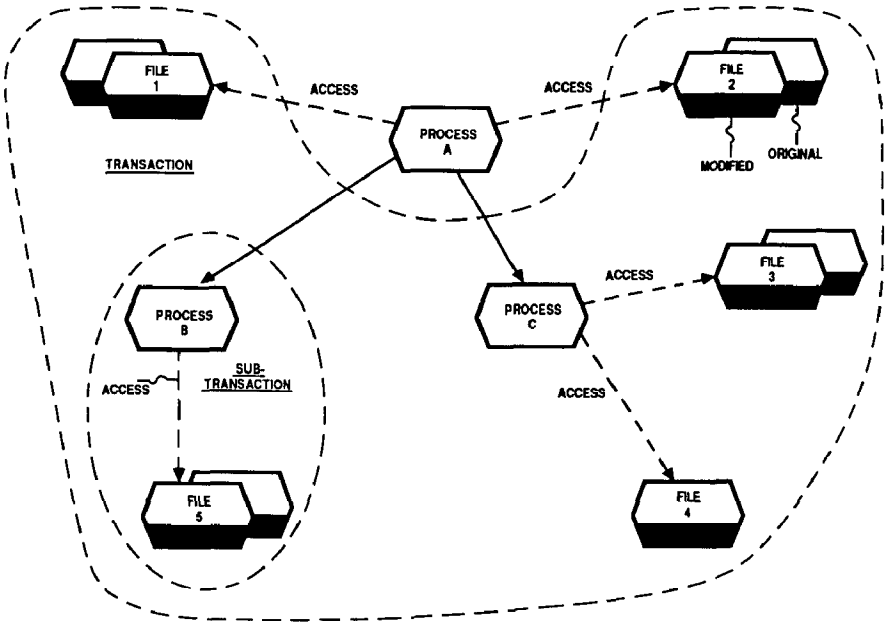
The mechanism allows groupings of users, so that access rights can be granted on a "wholesale" basis, i.e., to a specific group, irrespective of its members, rather than to each member individually. Groups can be hierarchically composed from subgroups, which then are "permanent members" of the enclosing group. As the figure shows, Group nodes can also have Potential_Member relationships to other group nodes. "Potential" is used because users belonging to a potential member subgroup can choose at any point in time whether or not they are to be considered members of the group.

As shown, users have a Default_Role relationship to a group of which they are normally considered members. These default roles are implicitly assumed at login; a

corresponding Adopted_Role relationship is created and is inherited by processes that the user starts. Processes may also be given Adopted_Role relationships to groups of which the user is a Potential_Member or to groups of which the executable code file is a member. The latter allows imposing restrictions that certain files can only be accessed through the services of a specific (group of) tools.

All nodes in the data base (not only file nodes, as shown in the figure) have access relationships to Group nodes. The Grant attribute on these relationships defines the kind of access allowed processes with Adopted_Role relationships to the Group or any Group that is a permanent member of the respective Group. The granularity of access rights lets the user specify different rights regarding attributes, relationships and contents; such rights distinguish reading, writing, appending, executing, and changing access rights. Access rights may also be user-defined and conditional: it may take a certain user-defined access right for a real access right to be granted. Since access rights are derived from the combination of all applicable access relationships, these conditional rights can ensure that only specific tools, or users in specific groups can perform certain operations on a node. The NEWMAIL and READMAIL rights in the figure are examples of user-defined rights, which ensure that users can append to mail files of other users via the mailer program, without otherwise being given any real access rights to those mail files.

Transactions



The CAIS support *transactions* at two levels of granularity: an entire process can be run as a transaction, so that all nodes modified by the process are included in the transactions, and processes can start transactions in which nodes are explicitly included. A transaction can be either *committed* or *aborted*. If aborted, the effects of all eligible operations that modified the included nodes or their contents are as if they never took place (note that not all operations are eligible: for example, writing to a printer cannot be undone.) In the figure, process A has started a transaction and included the nodes for Files 1 and 2.

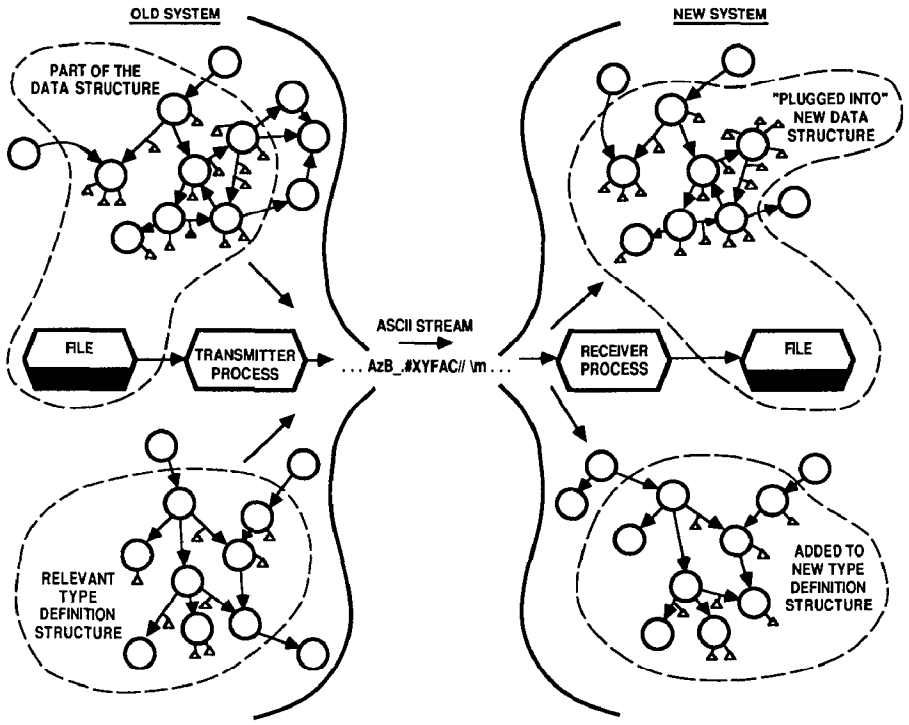
The explicit selection of nodes to be included in transactions, as contrasted to an implicit inclusion of all nodes modified between a "begin transaction" and an "end transaction" primitive, is a necessity when a process can have multiple execution threads (i.e., tasks) calling upon these primitives.

A process may run any number of transactions concurrently on different execution threads and even on a single execution thread. Nodes included in a transaction are properly locked against access from outside the respective transaction.

Also, transactions may be specified as being nested within other transactions. In this case, the effects on nodes included in the subtransaction are not available to the enclosing transaction until the subtransaction is committed. Upon such commitment, the effects are made available to the enclosing transaction, but the affected nodes remain included in the enclosing transaction. If an enclosing transaction is aborted, all on-going subtransactions are aborted and the effects of committed subtransactions are undone.

The transaction mechanism, the type structure, attribute monitors, and discretionary access control are expected to work together to support any desired level of *strictness* and *integrity* in the data structure. A large project with very tight time and budget constraints and a relatively junior programming staff might define an extremely rigid structure that restricts the things the programmers can do to a very narrow range. On the other hand, an experienced programmer working on a CAIS workstation entirely under his own control will be able to tailor the environment to any desired degree.

Movement of Data Between CAIS Systems



One of the main DoD requirements is to be able to move entire projects and parts thereof from one CAIS system to another, even when the two are implemented on different processors and different host operating systems. The common approach to requirements of this kind is to define an ASCII stream Common External Form (CEF) into which the data to be moved can be translated and from which it can be recreated.

The fact that almost everything in the system has an associated type definition makes it possible to translate the data structure into CEF. The user designates an arbitrary piece of the data structure for translation. The included items are translated into a stream of ASCII characters, transmitted to the other system, and used to re-create the structure.

Of course, the type definitions that apply to the data to be moved must be used to translate into CEF. They must also be used to translate from CEF on the new system and to access the moved data forever after. Therefore it is necessary to determine the relevant parts of the type definition structure, convert it to CEF, and transmit it before the designated data in the ASCII stream.

The one part of the data structure that the system cannot translate into CEF is the contents of File nodes. The person who defines a new node type with contents must also define *transmitter* and *receiver* programs (if the contents of the nodes are to be movable). The transmitter is invoked during translation to CEF to produce an ASCII stream of the contents, as shown in the left center of the figure. The receiver is invoked on the new machine to re-create the contents in the internal form of the new processor. Note that the receiver program, probably written in portable Ada, must be available for execution on the new machine; this could involve moving its source code, compiling, and linking it prior to moving any nodes.

Conclusion

The CAIS draws on the current state-of-the-art in SDEs. It may be unique in combining so many mechanisms (the ERA model, transactions, object typing, access control and security, a separate naming mechanism, a common external form, etc.) smoothly into an integrated whole. The closest equivalent of which the authors are aware is the PCTE+, and related work going on in Europe. The two projects have several contributors in common and have both used the RAC as a requirements source. Indeed, discussions between the two groups are underway and may well lead to a "merger" of some kind.

The CAIS is a large and complex system; much work remains to be done to determine if it can be implemented with sufficient performance on a variety of machines, including those with host operating systems and as an operating system in its own right, on "bare machines." We need to implement a great many tools and toolsets to make sure that CAIS provides all necessary facilities. (There is a bit of a "chicken-and-egg" problem here.) Beyond implementation, we need to accumulate a great deal of experience in using the system to do projects from the very small to the very large. It may be that the CAIS becomes most valuable over the full life-cycle of very large projects, meaning that a proper judgment of its worth will not be possible for many years.

References

[CAIS86] *Military Standard Common APSE Interface Set, United States Department of Defense, DOD-STD-1838,* 9 Oct 1986

[CAIS87] *CAIS Reader's Guide for DOD-STD-1838,* Institute for Defense Analyses, 14 Aug 1987

[MUNCK88] Munck, Robert, **Why Strong Typing was added to DOD-STD-1838, The Common APSE Interface Set,** *Proceedings of the Sixth Annual Conference on Ada Technology,* Washington, 15 March 1988

[RAC86] *DoD Requirements and Design Criteria for the Common APSE Interface Set (CAIS),* KAPSE Interface Team, Ada Joint Program Office, 4 Oct 1986

[RAT87] *Rationale for the DoD Requirements and Design Criteria for the Common APSE Interface Set (CAIS),* KAPSE Interface Team, Ada Joint Program Office, 18 Nov 1987

[RAT88] *Rationale for DOD-STD-1838 (CAIS),* Draft Version, Institute for Defense Analyses, 14 July 1988

[STONEMAN80] *DoD Requirements for Ada Programming Support Environments,* "STONEMAN," Feb 1980