Annual Report

August 1991

Compiling Knowledge-Based Systems Specified in KEE to Ada

NASA/Marshall Space Flight Center
Contract NAS 8-38488

IntelliCorp Inc.
1975 El Camino Real West
Mountain View, California 94040

Robert E. Filman, Principal Investigator
Roy D. Feldman, Scientist

(NASA-CR-190215) COMPILING KNOWLEDGE-BASED SYSTEMS SPECIFIED IN KEE TO Ada Annual Report, Jul. 1990 - Jun. 1991 (Intellicorp) 62 p CSCL 09B

N92-24538

Unclas

G3/61 0084336

Table of Contents

1. Overview

In this paper we report on the first year of the PrkAda project. Our aim is to develop a system for delivering Artificial Intelligence applications developed in the ProKappa system in a pure-Ada environment. We discuss the goals of the project, describe in detail the ProKappa core and ProTalk programming language; describe the current status of our implementation, with examples; discuss the limitations and restrictions of the current system; and describe the development of Ada-language message handlers in the ProKappa environment.

2. Goals

To quote one of our favorite books:

> *The United States Department of Defense (DoD) is a major consumer of software. Like many computer users, the Defense Department is having a software crisis. One trouble centers on the programming Babel—the department's systems are written in too many different languages. This problem is particularly acute for applications involving embedded systems—computers that are part of larger, noncomputer systems, such as the computers in the navigation systems of aircraft. Since timing and machine dependence are often critical in embedded systems, programs for such systems are often baroque and idiosyncratic. Concerned about the proliferation of assembly and programming languages in embedded systems, the DoD decided in 1974 that it wanted all future programs for these systems written in a single language. It began an effort to develop a standard language.*
>
> *Typical embedded systems include several communicating computers. These systems must provide real-time response; they need to react to events as they are happening. It is inappropriate for an aircraft navigational system to deduce how to avoid a mountain three minutes after the crash (in the unlikely event that the on-board computers are still functioning three minutes after the crash). A programming language for embedded systems must include mechanisms to refer to the duration of an event and to interrupt the system if a response has been delayed. Thus, primary requirements are facilities for exception handling, multi- and distributed processing, and real-time control. Since the standard is a programming language, the usual other slogans of modern software engineering apply. That is, the language must support the writing of programs that are reliable, easily modified, efficient, machine-independent, and formally describable. A request for proposals produced 15 preliminary language designs. The Defense Department chose four of these for further development. After a two-year competition, it selected a winner. This language was christened "Ada" in honor of Ada Augusta, Countess of Lovelace, a co-worker of Babbage and the first programmer. [Filman84, pp. 201--202]*

Ada has matured. There are currently many commercially available compilers for Ada systems. Ada is now required for much DOD, NASA, and other government programming. But a language developed primarily for embedded and real-time systems is awkward for developing Artificial Intelligence (AI) applications—Ada lacks many of the facilities which have eased the task of constructing AI systems. (As we shall see, Ada has certain properties that also make it awkward for deliverying AI applications—alleviating these misfortunes being a large component of this work.) The goal of this work, therefore, is to develop a system for Ada-language delivery of AI applications. That is, we propose that the user develop his or her system in a traditional, flexible AI environment (ProKappa). This includes creating a knowledge base of objects, a set of Ada-language message handlers, and a collection of rules and code in the ProKappa language ProTalk. Using the compilation tools developed in this work, the user's ProTalk can then be translated to Ada. This Ada can then be linked with our Ada-language object library and the user's message code and embedding system. The object library can, in the course of the application, read and create ProKappa knowledge bases. The net result is an AI application delivered in a pure-Ada environment. We illustrate this process in Figure 1.

From an applications point-of-view, AI as a technology has arisen because, using conventional technologies, it has proven difficult to build programs that solve certain classes of problems. These problems are often characterized by an irregular structure, by the necessity of creating complex data structures and applying semantically rich interpretations to these data structures, by the usefulness of a full library of data structure manipulation routines and an environment that can manipulate, coherently present and easily understand such structures. Often these mechanisms are packaged as knowledge-based systems (KBS) development tools. Traditional AI KBS tools provide:

Objects. Objects represent the elements of the domain of discussion. Objects have slots that describe their properties, can be arranged in class/instance hierarchies, inherit values along these hierarchies, invoke behavior on slot access and modification, and compute through messages.

Rules. Rules are a pattern/action, nondeterministic form of programming. They ease the programming task by enabling the encapsulation and quantumization of domain knowledge, and by freeing the programmer from explicit control decisions. (On the other hand, rules also complicate the programming process by encapsulating and quantizing domain knowledge and making the control decisions inaccessible to the programmer.)

Graphic development environment (GDE). A GDE provides the developer of a KBS with tools for understanding and modifying knowledge base structure and program behavior. Examples of GDE facilities can include "graphers" for presenting inheritance and other relationships as node/arc graphs, tabular or display formats for presenting and modifying object/slot values, and stepping program debuggers.

Application graphics toolkits (AGT). Application graphic toolkits are elements provided the application developer to aid in creating the end-user application graphics. Examples of such elements range from simple value displayers and single-choice, pop-up menus to the widgets of an X-windows toolkit.

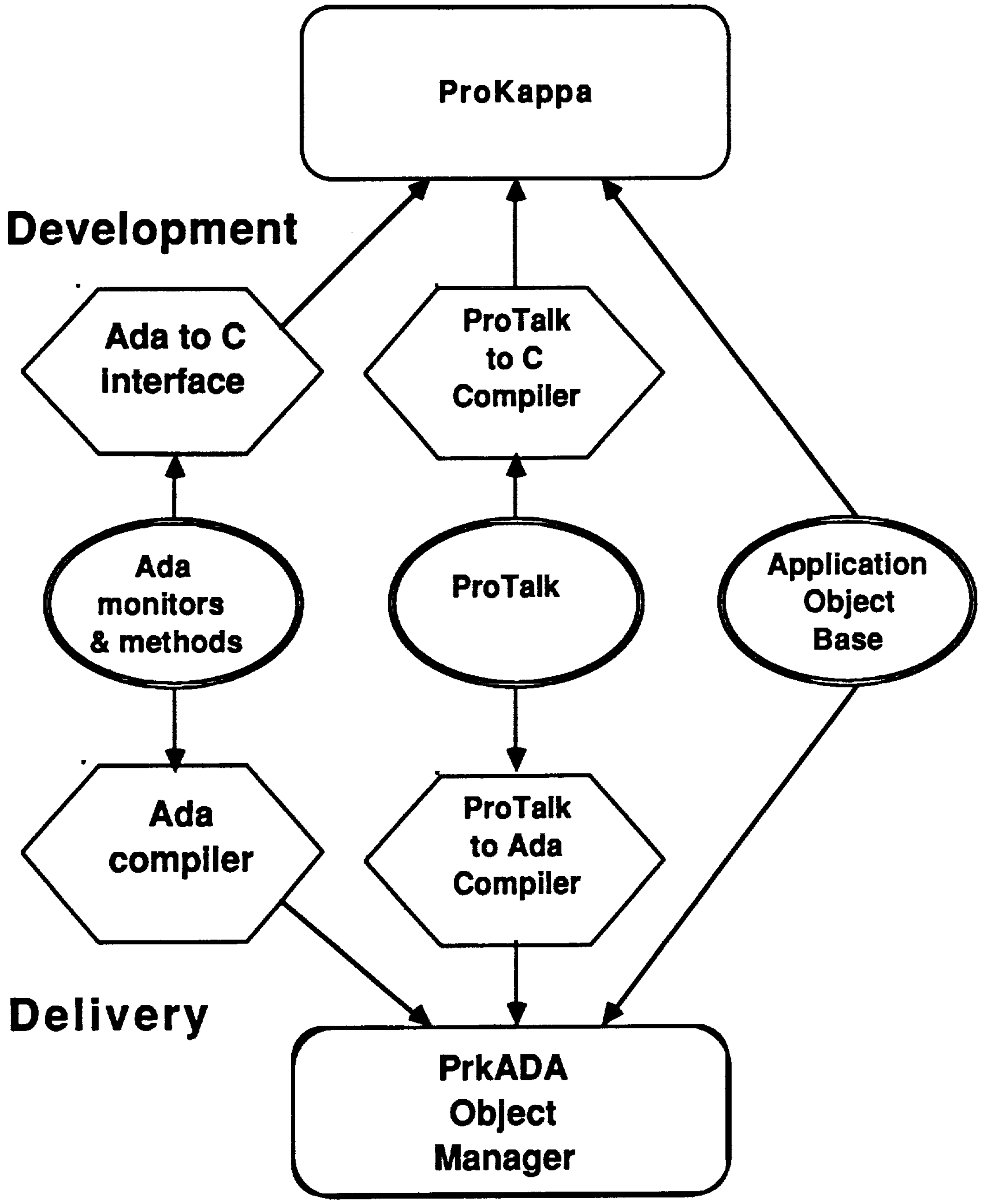

Figure 1. The development and delivery environments.

The earliest, most prominent, and most powerful commercial KBS development tools were written in Lisp. Examples of such systems include KEE [Fikes85], ART, and KnowledgeCraft. These systems built on the native Lisp environment to provide symbolic programming, automatic storage management (garbage collection) and an extensive set of native graphic, symbolic debugging tools. Unfortunately, such Lisp environments proved less than complete commercial successes. Problems faced by these products include dependence on an unusual programming language, a lack of connectivity to other systems, the requirement of a large run-time environment, and the difficulty of doing realtime programming with most implementations of garbage collection. These limitations restricted most KBS implementations to be either purely laboratory experiments or relatively stand-alone applications.

Currently, there is a trend towards the development of KBS tools in "more conventional" languages. Most such efforts are C-based. C seems to have emerge as the lingua franca of the programming universe. C has the advantage of being small, having an easily implemented compiler, of (consequently) running on almost all platforms, and of being familiar to many programmers. C is (except for the more obscure features) easily learned. C is flexible, allowing as it does access to most of the primitive machine operations, addresses of functions, and the run-time stack. On the other hand, C has several disadvantages: it is relatively unstructured, has few built-in language features to deal with the problems of building large systems, is somewhat non-standardized (and thus nontrivial to port), is basically unreadable and is difficult to maintain. Our favorite such C-based KBS tool is ProKappa.

3. ProKappa

ProKappa is IntelliCorp's C-based KBS development and delivery tool. ProKappa was first released in late 1990; it is still undergoing some evolution (which makes building a delivery environment for it somewhat more of a challenge.) As of this writing, the ProKappa development environment runs on Sun, IBM, and Hewlett-Packard UNIX-based workstations. ProKappa has the following major components:

Substrate. The substrate provides facilities corresponding to a Lisp system: datatype definition, creation, list utilities (e.g., length and print), and garbage collection.

Object manager. The object manager provides the object (frame) system, including inheritance and access/modification demons.

Rule system. The ProKappa rule system is called ProTalk. It allows the intermixing of rule-based and conventional, imperative styles of programming.

Developer interface. ProKappa provides both a graphic developer interface and Sabre-C, a symbolic "read-eval-print" loop for C.

User interface toolkit. ProKappa provides a number of predefined images, specified by objects, an object-based dialog box facility, and access to X/Motif.

Database access. ProKappa includes a module for accessing SQL databases, moving data between the database and object system.

As we are creating a delivery system, we deal with only the first three of these in this work. In general, we speak of the substrate and object manager together as the *core*.

3.1 Substrate

The substrate corresponds to the Lisp level of KEE-like environments. The ProKappa substrate provides several different primitive datatypes, which collectively form the what is officially called the type ProType. In our more mathematical descriptions below, we call this set the "domain." Most important subtypes of ProType are:

Symbols. Symbols are like the symbols of Lisp. Symbols have properties (a property list) and print names. In ProKappa, there is a single symbol table (unlike the packages of Common Lisp). All symbols are on this table; ProKappa does not support uninterned or generated symbols. There are functions for taking a string and returning the corresponding symbol. Symbols are permanent; they are not garbage collected and cannot be explicitly deleted.

Cons cells. These correspond to Lisp's cons cells. A cons cell has a car and cdr field, each of which can hold a ProType. There is a cons operation for creating cells, and rplac operations for modifying cell contents. Cons cells (and arrays, below) are garbage collected.

Arrays. These are dynamically created, zero-based, and garbage collected. Each array cell can hold any element of the domain. Arrays keep track of how large they are. That is, an array has a size (below its maximum size) and there are explict operations for changing that size.

Numbers. ProKappa supports three kinds of numbers: integers, floats, and double floats.

Booleans. True and false.

Strings. Sequences of characters.

Methods. Methods are pointers to functions. A method can be a slot value on an object; sending a message to that object selected by that slot applies that function to the remaining arguments of the message. Ada does not have function-valued datatypes; as we discuss below, this presents a challenge for our Ada core.

Objects. Objects represent the elements of the application domain of discourse. Objects have many fields, including slots (and facets of slots). These slots and facets have values, which are drawn from the domain. We discuss objects in detail below.

The substrate also supports several other primitive data types. Many of these represent internal system structures, and perhaps should not have been documented at the user level.

The ProKappa substrate also provides automatic storage management (garbage collection, or GC). That is, AI systems usually revolve around allocating elements from the heap, and passing and storing many pointers to such elements. At any point in the execution of the program, there are certain live "roots"—elements identified by name in some active portion of the program. These are basically (1) global variables and structures that include pointers, and (2) variables and structures allocated on the stack (the locals and parameters of the currently active procedures, back to the main calling program). These roots include pointers to other heap-allocated structures, which include pointers to other heap structures, and so forth. The closure of this "pointing to" relationship over the roots constitutes the live storage. As long as (1) we can identify which part of each structure is a pointer, and which, constant data (e.g., numbers), and (2) the program does not execute any pointer conversion or pointer arithmetic operation (e.g., using unchecked_conversion on a pointer), then all other heap allocated storage is inaccessible—it is garbage; its space can be reused. Having a program mechanism that collects garbage is useful, because a garbage collector enables a large class of programs that would otherwise run out of space to continue indefinitely. It saves the programmer the difficult (if not impossible) diligence of knowing when a shared structure is no longer in use, concomitantly enables considerable sharing instead of copying, and cures a source of insidious bugs.

There are two major themes of garbage collection algorithms, reference counts and pointer following. Reference count mechanisms keep track, for each cell, the number of active pointers to it. Every time a pointer is checked, these reference counts must be updated. A cell whose reference count goes to zero is garbage, and may be added to the free list. Reference count mechanisms require capturing every pointer modification, have the advantage of spreading out the effort of garbage collection through a programs execution, and the disadvantages of requiring space in each cell for the reference count and of being unable to collect circular structures that are nevertheless garbage. (That is, as illustrated in Figure 2, the given cells, with no external pointers, are garbage, but each still has a positive reference count.)

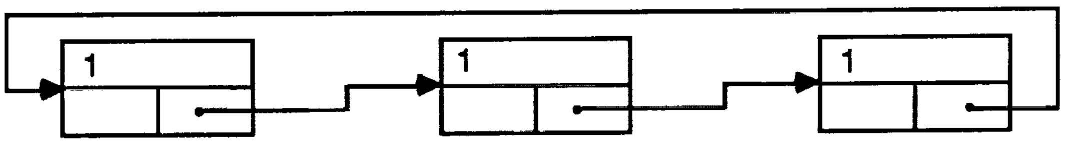

Figure 2. Circular garbage

Pointer following algorithms allocate (during garbage collection) a "mark bit" for each cell or structure. The algorithm consists of (1) turning off the mark bit of all cells, (2) starting at the roots, marking each cell by following pointers (sweeping). It is unneccesary to follow the pointers of cell that is already marked. When this process is complete, all cells that are still unmarked are garbage. Mark and sweep algorithms have the advantage of finding all garbage and allowing a simpler assignment statement, but the disadvantage, in this unmodified form, of requiring a pause in the overall program activity (which can impact the real-time or regular behavior of the system.) Mark and sweep also requires access to the program stack, which is allowed in C but not possible in Ada. There are versions of pointer following algorithms that avoid the use of the stack during garbage collection and which can be run concurrently with the main process ("real-time garabage collection").

ProKappa uses the latter approach. ProKappa reserves the lowest three bits of each data item as a tag, using this to discriminate between pointers and non-pointers and among the varieties of each. (This can potentially cause some "false positives"— non-pointer data that is confused with a pointer, resulting in uncollected garbage and extra work for the garbage collector, but cannot result in collecting non-garbage.) In Ada, lacking both stack access and reliable access to the dereferencing of pointers, more convoluted mechanisms are required. We discuss these below.

3.2 Objects

In this section we attempt a relatively more formal specification of the ProKappa object system. We have taken the liberty of modifying the names of functions and predicates in this more formal specification. We are really describing a collection of programs that allocate storage locations and, in the course of program execution, modify these locations. Hence, the veneer of predicate calculus semantics must be understood in the context of the semantics of assignment operations of executing programs. Our logic describes static truths—things true at a single point in program execution. Since programs actually modify data, a statement may be true at one instant and false later, much as a variable may take changing values in the course of program execution. Sometimes, a particular operation may have a "universally quantified consequence," which should be understood to mean that it modifies several locations. Additionally, the modification of a location may have other, to be specified, consequences.

The object system is built on the substrate. ProKappa provides two kinds of objects, classes and instances. Every object is either a class or an instance, but not both. (This constrasts with KEE, where an object could be both a class and an instance of another class). That is,

forall x. object(x) -> (class(x) /= instance(x))

ProKappa provides the relations subclass (class, class) and element-of (instance, class). The inverses of these relations are superclass and class. For example, we can have a class of sensors, with a subclass of electrical-sensors. AC-Sensor22 can be an instance of electrical-sensors. Subclass and element-of (and superclass and class) are many-to-many relations—for example, an instance can be an element-of many classes; a class can have many subclasses. (The parents of an object are explicitly ordered; more formally, there is a map from the parents of an object to a contiguous set of the integers starting at one. Less formally, the parents of an object are kept in a list, and sometimes the order of the parents in this list matters.) Collectively, an object which is a subclass or element-of another object is a child of that object. The inverse relation of child is parent. Correspondingly, we use the terms ancestor and descendant to express the closure over the parent and child relations. Subclass cycles are illegal; that is, we cannot have

subclass(A0, A1) & subclass (A1, A2) & ... & subclass (An, A0)

Certain instances are applications and modules. There is a relation, in, such that every object is in a unique application or module. That is,

forall x, object (x) -> exists! m. ((application (m) or module(m)) & in(x, m)

Modules are only in applications; all applications are in a special "system application," which is in itself. That is,

forall x, y. module (x) & in(x, y) -> application (y)
forall x. application (x) -> in (x, system_application@)
in (system_application@, system_application@)

Objects are either named or are anonymous. Names are symbols. There is a function name-of (object) |-> symbol that returns the name of an object. Unfortunately, in ProKappa the name space is universal—there can be only one object of a given name in the system. (This mistake will be corrected in latter versions.) More formally

forall x, y. named-object(x) -> (name(x) = name(y) -> x = y)

Like symbols, objects have properties. That is, there are functions set-property (object, symbol, domain), get-property (object, symbol), and remove-property on objects. These operations behave with assignment semantics.

Objects have slots. Abstractly, a slot is a binary relation on objects and the domain. That is, defining a slot on a class of objects is creating a new relation on that class (and, under most circumstances, its descendants) whose second parameter is on the domain. (Of course, mathematically, all relations exist over every object, and are just false or undefined for lack of other information. However, structurally, slot creation extends objects; pragmatically, most predefined ProKappa functions error if invoked on objects without the specified slot.) As a corollary of the concrete realization of slots, every slot in an object has a unique name, drawn from the set of symbols. Thus, it is more correct to view slots as a collection of binary and ternary relations: has-slot (object, symbol), and has-value (object, symbol, domain). It is also useful to speak of slots concretely, as, for example, "the slot S." This should be understood as shorthand for "the slot S in object O."

"In the default case," slots inherit over the subclass and element-of relation. That is, (with exceptions to be described below) if class C has slot S, then every subclass and instance of C also has slot S. More specifically, every slot has a slot type, one of default, subclass, or own. Default slots inherit to subclasses and instances. Subclass slots inherit only to subclasses. Own slots do not inherit at all.

Every slot also has an inheritance role, one of: single-value-no, multi-value-no, single-value-override, multi-value-override, method, single-value-initial, multi-value-initial, self-first-union, self-last-union, and monitor. (This last role applies only to facets, discussed below.) In general, inheritance roles split into single-valued roles and multi-valued roles. A single-valued role implies that at most one element of the domain can be in this slot at any time; that is, the slot is functional. The system errors on an attempt to add more elements of a single-valued slot. Method inheritance (for inheriting functions) is single-valued; monitor inheritance, multiple-valued.

Default slots inherit to subclasses and instances, subclass slots only to subclasses, and own slots, not at all. We call the slot type and inheritance role of a slot it's signature. It is an error to have an object with two parents with slots of the same name but different signatures.

```
forall p, c, s. has-slot (p, s) &
                  ((slot-type (p, s) = default  & parent (p, c)) or
                  (Slot-type (p, s) = subclass & superclass (p, c))) ==
                        inherits (p, c s)

forall p, c, s. inherits (p, c, s) -> has-slot (c, s)

forall p, c, s, x. inherits (p, c, s) & slot-type (p, s) = x ->
                        slot-type (c, s) = x

forall p, c, s, x. inherits (p, c, s) & inheritance-role (p, s) = x ->
                          inheritance-role (c, s) = x

forall p1, p2, c, s, x. ~ (inherits (p1, c, s) &
                          inherits (p2, c, s) &
                            (slot-type (p1, s) /= slot-type (p2, s) or
                            inheritance-role (p1, s) /=
                                      inheritance-role (p2, s)))
```

Slots originate only in classes. That is, no instance object has any slot that is not a default slot of (at least) one of its parents. More specifically

```
forall c, s. instance (c) & has-slot (c, s) -> exists p. inherits (p, c, s)
```

Slots in objects have values drawn from the domain. Single-valued slots have at most one value. The values of a multiple-value slot are an ordered set—the values have an order, but values do not repeat. For the sake of further discussion, we introduce the notion of local-value. A value is a local-value of an object, slot if it has been explicitly asserted that that object, slot has that value. (There is effectively a constant called "unknown," (which we designate as "?") which is the default local-value of all slots.)

The combined-value (or, more simply, the value) of an object, slot is a function of (1) the local-value of the object, slot; (2) the inheritance role of that object, slot; (3) the combined-values of the slot's parents. The intended semantics, by inheritance role, is

No inheritance: The local value is the combined value.

Override inheritance: The local value is the combined value if it is not unknown, otherwise, the values of the first parent in the parent order with any values (non-unknown) is used.

Initial inheritance: If the local value is unknown, the semantics are the same as override inheritance. With changing from unknown to a local value, the combined value is made local. With an already existing local value, that local value.

Self-first-union and self-last-union: These roles combine the values of all parents into a set, putting the local values first (last).

Let ***exists-first*** be a quantifier that selects the first element of a set that satisfies a predicate. Ignoring the single-value/multiple-value issue, we get

```
forall c, s x. inheritance-role (c, s) = no ->
        forall x. combined-value (c, s, x) == local-value (c, s, x)

forall c, s.   inheritance-role (c, s) = override ->
       (((exists x. local-value (c, s, x) &  x ~= ?) ->
              (forall y. local-value (c, s, y) == combined-value (c, s, y))) &
       (((forall x. local-value (c, s, x) -> x = ?) ->
              (exists-first p in parents (c) such that
                 inherits (p, c, s) &
                  exists y. combined-value (p, s, y) ->
                     forall z. combined-value (c, s, z) ==
                            combined-value (p, s, z)))))

forall c, s.   inheritance-role (c, s) = self-first-union ->
          (forall x. combined-value (c, s, x) ==
              (local-value (c, s, x) or
              (exists p. parent (p, c) &
                     inherits (p, c, s) &
                       combined-value (p, s, x))))
```

Initial inheritance is like override, except that changing the local value of an initial slot from unknown causes the current combined value to be installed as the local value. Self-first-union and self-last-union differ in the order of the index of the values (local values first or last). Method inheritance has the same semantics as override inheritance, except monitors (disucssed below) are not run. Monitor inheritance is like union, except values are indexed by their priority order, highest first.

Slots have facets. That is, more formally, there is a relation has-facet (object, symbol, symbol), and a relation facet-value (object, symbol, symbol, domain). Facets inherit with their containing slot. Thus,

```
forall c, s, f. inherits (c, s) == facet-inherits (c, s, f)
```

Facets have inheritance roles. In contrast with slots, it is legal to introduce facets at the instance level. Facets hold values. Conceptually, facets are for additional "annotational" information on the slots. More formally, we could extend the set of possible slots to include a "facet-name" type, provide an operator that takes two symbols and creates a facet-name, and provide similiar axioms, with exceptions such as the slight differences in inheritance roles, lack of monitors, and the ability to introduce facets on any object.

There exist functions for dynamically creating new classes and instances, new slots in classes, and new facets on on objects. Slots and facets may be deleted; "deleting" an object marks it as deleted (unusable for other operations) but does not reclaim its storage. (Hence, it still exists as an element of the domain.)

There are functions for testing if a slot or facet exists on an object and for retrieving the elements of the signature on the slot and facet. The signature of a slot or facet can be changed only at the "originating" object—an object with the slot whose parents do not have the slot. (In the case of multiple inheritance, this could cause a signature-conflict error.)

There also exist functions for dynamically changing the parents of an object. This may cause some slots on the object to cease to exist. When a new slot is created in a class, (module the slot type) that slot is inherited to the descendants of the class. There are functions for setting, adding-to, removing from, and retrieving the values of slots and facets. The semantics and behavior of these functions (at least with respect to slots) is modified by the monitor mechanism,

3.3 Monitors

Monitors are objects designed to "watch" slots, acting on changes and retrivals. Monitors can act on slot value modification, slot value retrieval, and on associating with (attaching) or disassociating from (detaching) a slot. Monitors objects have three significant slots, "action," "attached," and "detached." The values of these slots are functions. Monitors also have a "level," a priority, and several flags.

There are two kinds of monitors, WhenNeeded monitors and WhenChanged monitors. WhenChanged monitors split further into two varieties, BeforeChanged monitors and AfterChanged monitors. Semantically, WhenNeeded monitors run on value retrieval; WhenChanged, on value modification. BeforeChanged monitors run before a slot's value is changed, and can modify what the change will be; AfterChanged monitors run after the slot's value is changed.

Monitors can be attached to slots. Effectively, a monitor is stored on a WhenNeeded or WhenChanged facet of that slot. This implies that monitors can be inherited. The act of attaching a monitor (even if that attachment comes through inheritance) causes the attach method of the monitor (if any) to be run. The act of detaching a monitor correspondingly causes the detach method of the monitor to be run.

All monitor functions take as a parameter a "monitor info" data structure, which has fields describing the object and slot on which the monitor is running, the monitor object itself, whether the inheritance role of the slot is single or multiple-valued, and the level, flags, and priority of the monitor. When there are several monitors attached to a particular slot, they run in priority order, from highest to lowest. Associated with storage and retrieval operations is a numeric "level" (which defaults to the value of a global variable). Monitors whose level is above this value are not run (suppressed.)

More specifically, when a slot value modification operation (setting, adding or removing values) is done on a slot that does not have any WhenChanged monitors, the local value is computed and the inheritance role is used to determine the new combined value.

When a slot with WhenChanged monitors is modified, the system computes the set of BeforeChanged monitors whose level is greater than or equal to the level specified in the modification operation. It orders this set by priority. It then applies the action of these monitors in turn to the newly added values, newly removed values, and monitor information. Each function computes new sets of changed values, which are then given to the next monitor in the priority sequence. After the last BeforeChanged monitor, the added and removed values and the inheritance role are used to determine the new slot values. Then the AfterChanged monitors run, once again filtered by level and in priority order. These monitors get the old and new values. They have no direct effect on the slot value. However, by executing the appropriate side effects, such monitors can start the whole cycle again. Various flags on monitors determine if the monitor is run on a set/add operation that doesn't actually modify the value in the slot (because of redundency), on monitor attachment, on application load, and so forth.

WhenNeeded monitors run on value retrieval. There is a similar filtering by level and ordering by priority. WhenNeeded monitors can modify the value apparently retrieved. Facets do not have monitors.

Monitors are useful for activities such as checking the type of new values, coercing new values to an appropriate type, recording statistics on slot usage, maintaining relationships among slots, keeping a graphic display synchronized with a slot value, debugging, performing a functional translation of values, and making complex computations (such as running a model) appear to be simple value retrieval.

3.4 Object-oriented programming

Object-oriented programming (OOP) centers on the ideas of (1) identifying particular individuals (data structures) as objects, and (2) providing a uniform interface to behavior that can vary by individual. True object-oriented programming allows dynamic binding: the behavior associated with a particular program variable is not known at compilation. Of course, in ProKappa, the objects of OOP are the knowledge-base objects. Varying behavior is achieved by a simple trick: we make the value of a slot be a function, and interpret a message to an object indexed by that slot name as the application of the function in that slot to the object, slot, and remaining arguments of the message. Since this is C, such functions are in the global context; they run without the benefit of an Ada or Lisp-like enclosing environment. (In Ada, as we shall see, functions are not objects and this becomes more difficult to achieve.) Unlike KEE, ProKappa lacks method combinators; unlike SmallTalk, there is no primitive send-super, though one could be written at the user level. Because message handlers are stored as slot values, methods inherit through the object hierarchy.

Object-oriented programming in AI systems contrasts with more conventional programming language notions of OOP—AI approaches provide greater flexibility (for example, the ability for objects to dynamically change message handlers or even dynamically change class). ProKappa accomplishes this by implementing all objects uniformly and providing specific accessing functions to the parts of an object. With the appropriate indirection, even this can be avoided [Filman86].

3.5 Miscellaneous facilities

ProKappa provides procedures for saving the state of an object system and reloading saved states. In this respect, applications and modules are modular—they can be saved and reloaded independently. ProKappa also has facilities for source and version management—the equivalent of Unix make files for remembering the components of an application.

The hierarchical nature of the ProKappa type system constrasts with the flatter structures of a conventional language like C. As C lacks mechanisms such as Ada's overloading, C written for ProKappa turns out to be quite dense in casts. To alleviate this problem, ProKappa includes a Happy C preprocessor, which takes a C program and inserts the appropriate casts (and coercions) for arguments. Happy C also includes a quoting character that allows the simple specification of constant symbols, lists, and arrays.

3.6 ProTalk

A fundamental result of programming language design is that language constructs should express intent. Often in AI, the intent is to express quantified statements of a general form of "when this happens, this should follow." In classical AI systems, this has been realized with rule based programming. (Rules are used frequently enough in AI that some confuse AI with rule-based programming.) However, often intent in AI matches conventional programming structures—sequencing, conditionals and iteration. This section describes ProTalk, a ProKappa language that melds rule-based and imperative programming.

Conceptually, rule based systems are founded on pattern-action programming. A rule implements a pattern-action pair. When the pattern matches the situation, it is appropriate to execute the action. Rule languages leave open whether the pattern-actions express truth (when this is true, conclude the following) or programs (when this happens, do the following.)

A pattern-action programming language has two important characteristics that separate it from a conventional language. The first is the need to describe patterns. The second centers on the issue of conflict resolution: what to do when several patterns simulateously match. The semantics of a particular pattern-action language may range from requiring this choice to be made non-deterministically through specifying complex rules for ordering the rule selection. Whatever point is chosen on this continuum, this conflict resolution problem introduces considerable intellectual complexity to the programming process.

Thus, rule languages are a two-edge sword. They allow "atomization" of knowledge (the independent assertion of separate facts), the assertion of universally quantified statements, and free the programmer from concern for control structures and sequencing. However, they present a the non-deterministic semantics, introduce unanticipated interactions between elements of the atomized knowledge, require considerable effort in establishing context for each atomic knowledge element [Bachant89], and demand circumlocutions and idioms to obtain conventional control patterns such as loops and conditionals. Examples of rule languages include Prolog [Clocksin84] and OPS5 [Brownston85]; almost every AI tool has a rule language of some form.

Because rules have some of the mathematical semantics of if-then-else, rules can be run (chained) either in a "forward" or "backward" direction. Forward chaining requires following all the consequences of an assertion. That is, if we have rules that state

$$\text{if } A \text{ then } B, \qquad (1)$$

and

$$\text{if } B \text{ then } C, \qquad (2)$$

then a forward chaining system will, on the assertion of A, conclude B (using rule 1), then conclude C (using rule 2). OPS5 is example of forward chaining system.

Backward chaining involves reasoning from a consequence to the conditions that cause that consequence. Thus, we could backward chain with the rules given the question "is C true" to the conclusion that A's truth implies C's truth. Prolog is an example of a backward chaining system. Some systems allow mixed chaining— it is possible to invoke forward chaining while back chaining, and visa versa. Our course, our A, B, C example simplifies the problem, as most rule systems allow variables for the patterns A, B, and C, and much of the search involves finding bindings for the pattern variables that satisfy the rules.

Rule systems also differ on whether they are "always active" or "invoked." In always active systems, the rule-based consequences of any assertion that affects a rule are always followed. The advantage of always active systems is that since all knowledge-base assertions are noticed by the rule system, one can build more efficient algorithms for the rule mechanism (such as RETE [Forgy82]). Invoked systems require specific rule system invocation. Invoked systems require a programmatic control, but allow restricting search to only relevant rules. This results in more straightforward control structures and often, greater efficiency.

There are three primary ways of implementing rule languages: interpreters, compilation to a network, and compilation to an abstract machine. Interpretation (used, for example, in KEE) requires an engine that successively examines appropriate rules in turn and explicitly executes a rule coherently. This requires little additional storage, and allows invocations of subsets of all rules. However, it is not as directly efficient (in a raw-machine sense) as compilation mechanisms. The RETE mechanism compiles rules to a network, progressively advancing tokens through that network as portions of rules are matched. This is perhaps the most efficient way of implementing a collection of rules, but can have large space requirements, works only for forward-chaining, and does not readily lend itself to rule subsets. Abstract machine compilation effectively treats rule languages as would a conventional compiler, treating the sequence of instructions as requiring movement of data between locations. Abstract machines typically have primitives for pattern matching and storing the search context of the rule system. Compilation to the Warren Abstract Machine (WAM) is the standard strategy for implementing Prolog systems [Warren77].

The goal of ProTalk is to span the continuum between conventional languages and rule languages (and, in that process, to integrate with the frame system). That is, ProTalk seeks to let the programmer say in conventional constructs those things that are best say with conventional constructs, but to include built-in pattern matching and rules. ProTalk in this respect traces its intellectual roots to Planner [Hewitt71], which extended Lisp with pattern-matching and search constructs.

ProTalk relies on the following fundamental concepts:

Rules and functions. ProTalk subprograms are either functions (with parameters, returning values), forward chaining rules, or backward chaining rules. Rules are divided into condition and action parts; the condition is typically a series of expressions to be matched. In general, all ProTalk operators are available in any context.

Variables. Variables in ProTalk are neither declared or typed. (Typing is avoided as all are of type ProType.) Variables are initially unbound. Evaluating an expression containing unbound variables is effectively a request to find a binding that matches the variables. Subsequently (in that subprogram or rule), the thus bound variables take the value of this binding (until rebound or assigned.) ProTalk in this respect resembles Prolog. However, unlike Prolog, one cannot bind together two unbound variables.

Success and failure. Like ProLog, ProTalk incorporates a notion of success and failure. A ProTalk program is a series of statements. Each statement either succeeds or fails. Typical failures include the inability to find a binding for a variable or a false evaluation of a Boolean expression in a non-testing context. Failure backtracks to the last choice point (place where there were multiple ways of getting an answer) and considers the next choice. Success proceeds to next statement. In functions, choice points must be created explicitly (using the find operator). Rules have an implicit find before all statements; hence, rules may create many choice points. Unlike ProLog, in ProTalk one can mix conventional control constructs, such as assignment, if/then/else, for/while, iteration over lists and into accumulators, in functions and rules.

ProTalk and ProKappa. ProTalk is effectively connected to the core—there are primitive operations for changing/inquiring about class/member links, slot and facet values. (Unfortunately, the semantics of ProTalk operations can depend on the multiplicity of slot inheritance roles.)

The following is an example of a ProTalk function, `CslResetForRules`. It takes two arguments, `?self` and `?slot`. It starts by setting the `Pressure` slot of the object `H2Source` to the symbol `Normal`. It continues by looping through each child subclass of the object `Subsystem` (`?X`), looking in the `SubsystemProblem` slot of that child. It removes any value it finds. It then loops through all the descendants of the class `Components`, binding them successively to the variable `?Unit`. For those that are not in the specified list of exceptions, it sets the `value` slot of the `?Unit` to the symbol `Normal`. It concludes by running the rules in the set of `CoolingTestProcedureRules` to find values of the `FaultyComponent` slot of the object `Cs_1`, printing out a message for each.

```
function CslResetForRules (?self, ?slot)
{
    H2Source.Pressure = Normal;

    for ?X inlist all direct subclassof Subsystem;
    do
    {
        for {find ?X.SubsystemProblem        == ?sp; }
        do
        {
            ?X.SubsystemProblem              -== ?sp;
        }
    }

    /* Set Component Value      */
    for ?Unit inlist all instanceof Components;
    do
    {
        if Member (?Unit, `(E1@, E2@, E3@, E4@, E5@, E6@, I1@, W1@))
           == FALSE;
        then
            ?Unit.Value = Normal;
    }

    for ?component =
            find [ CoolingTestProcedureRules ] Cs_1.FaultyComponent;
        do {Print("\nA faulty component is ");
            Print(?component");
           }
}
```

This function is intended to be called with ?self and ?slot bound. Calling a function with unbound variables allows the function to act as a generator—successive calls, within the context of a search, produce new bindings for the unbound variables. When all bindings have been exhausted, the function call itself fails. ProTalk requires that all parameters to a functions must be bound on function exit.

The following is an example of a ProTalk rule—a backward chaining rule, to be more precise. It specifies an immediate rule class for the rule (`AllLowAll_EsShortRules`). (Rule classes can themselves be built into directed acyclic graphs, though rules do not correspond to particular objects in the object system.) This rule "means" that if a component (`?comp`) value is (the symbol) `high`, and a sensor of that component (`?sen`) has a `high Trend`, then the component's `ComponentFailureType` slot is to have the value `ReadsHigh` added to it, and the component's `FaultState` slot is to be set to `UncorrectedFault`. The function `RespondToHighSensor` is then run on component. Used in a backward chaining fashion, this rule can be understood to mean, "if one is looking for something with `ReadHigh` in its `ComponentFailureType` slot, or `UncorrectedFault` in its `FaultState` slot, then establish that that thing has `High` in its `Value` slot, and it's `sensor` slot contains an object that has `High` in its `Trend` slot." If a rule succeeds, the rule itself is run, causing execution of the function `RespondToHighSensor` on the component.

```
bcrule D1ReadsHighRule in AllLowAll_EsShortRules
{
  if:
      ?comp.Value   == High;
      ?comp.Sensor  == ?sen
      ?sen.Trend    == High;
  then:
      ?comp.ComponentFailureType +== ReadsHigh;
      ?comp.FaultState           = UncorrectedFault;
      RespondToHighSensor(?comp);
}
```

3.6.1 Syntactic detail

Having specified the intent and examples of ProTalk, we now descend to its specific syntactic detail.

ProTalk has constants that correspond to the constants of the subtypes of PrkType. In general, a free identifier corresponds to the symbol of that name; followed by an `@`, it denotes the object of that name. Symbols can also be backquoted (`` ` ``); backquoting a string creates a symbol with an arbitrary name. The dipthong `` `! `` denotes a method. The syntax for numbers, characters, and strings follows the usual C-language notation. Prefixing an identifier with `?` (e.g. `?foo`) creates a named variable; `?` by itself is an anonymous variable, in the Prolog sense. ProTalk freely coerces symbols to objects when a symbol is used in a context that requires an object.

Lists are created by backquoting as sequence in parentheses; commas separate list items, and `|` can be used for dotted-notation. For example `` `(Foo, Baz@, ?X | ?Y) `` is the equivalent to the effect of the Lisp evaluation of (cons symbol-foo (cons object-baz (cons variable-x variable-y)))) — i.e. `(Foo Baz@ ?X . ?Y)`. Arrays follow a similar notation, substituting square brackets for parentheses.

Expressions are created using operators and parentheses. Common operators such as + and < are included; additionally, the language has about two dozen operators with special meanings. The most elementary are those for accessing the object system; these are listed in Table 1. Such expressions can be nested by use of parentheses.

Slot value	{Object}.{slot}
Facet value	{Object}.{slot}..{facet}
Instance children of a class	`direct instanceof` {class}
Instance descendants of a class	`instanceof` {class}
Subclass children of a class	`direct subclassof` {class}
Subclass descendants of a class	`subclassof` {class}
Class parents of an instance	`direct classof` {instance}
Ancestors of an instance	`classof` {instance}
Class parents of a class	`direct superclassof` {class}
Class ancestors of a class	`superclassof` {class}

Table 1: Templactes for knowledge expressions.

Information in the object base can be modified by use of the assignment ("="), addition (+==) and deletion (-==) operators. Thus, `air_exchange.symptom +== ?problem` specifies that the value of the variable `problem` is to be added as a value of the slot `symptom` on the object `air_exchange`. `?problem = air_exchange.symptom` specifies that the variable `problem` is to be bound to a value of `air_exchange's symptom` slot. An expression which accesses the object base is called a knowledge expression (KE).

KEs can accept search modifiers. These are specified in Table 2. Note that some of these modifiers are deterministic while others are non-deterministic.

No modifier	Generates a single value; unknown if no value.	Deterministic
`All`	Generates a list of values; nil if none.	Deterministic
`Find1`	Generates a single value; fails if none.	Non-deterministic
`Find`	Generates one value at a time, acting as a generator. Fails when it runs out of values.	Non-deterministic

Table 2. Knowledge expression modifiers.

Non-deterministic operators can be modified by "`sum`", "`collect`" or "`count`" to sum, collect into a list, or count the instances that match them.

ProTalk has the usual complement of (C-syntax) binary relations: == (equality; match), >= ... != . The match operator, ==, can also be used for pattern-matching on lists (much like ProLog). For example, if `?first` and `?rest` are unbound and `?AllEs` is bound to a list, on executing

```
?AllEs == `( ?first | ?rest );
```

`?first` will be bound to the car of that list, and `?rest`, to its cdr. If, on the other hand, `?first` were also bound on execution, then either (1) `?rest` would be bound to the cdr of `?AllEs` if `?first` was the same as the car of `?AllEs`, or (2) the statement would fail.

ProTalk has conventional if/then and if/then/else statements. The success of an if/then/else statement is the success of the chosen branch; the success of an if/then statement with a true condition is the success of the then part; an if/then statement with a false condition always succeeds. (A case statement can be used as a compact syntax for repeated if/then/else's. For example,

```
if ?textitem == Low;
then
{
    if ?item > ?last;
    then
    {
        return (TRUE);
    }
}
else
```

```
    {
        return (IsIntermittent (?textrest, ?rest, ?last));
    }
```

ProTalk has an extensive collection of iteration operators, including iteration through the elements of a list or array (inlist and inarray), and numeric counter (from/to). For example,

```
for ?X inlist `(E1@, E2@, E3@, E4@, E5@, E6@);
do
{
     if ?Unit == ?X;
     then
          ?Temp = ?Value;
     else
          ?Temp = ?X.Value;
}
```

Iteration operators (or simply statements under a find) can be combined with accumulators to sum or count the items in a set, or build a list of the items of a set. For example,

```
for find ?total = count; ?filter.value == high
```

The bound operator tests to see if its variable argument is bound. The statement "bound inputs;" is an assertion that all the parameters of a function are bound. The return operator can be used to immediately return a value from a function.

Syntactic escapes exist for calling a library of ProTalk functions that correspond to the functions of the ProKappa core and for inserting C code directly into a ProTalk application. ProTalk also includes an extensive library of built-in functions, which correspond to most of the core functions and a small mathematical library.

A ruleset is a collection of rules which are to be run together. The declaration of a rule requires placing it in a single, direct ruleset. Rulesets can be subsets of other rulesets; rules are run with respect to a particular ruleset (or the current ruleset, if there is one.) Thus, by being in a ruleset that is a subset of several rulesets, a given rule can be in a multiple rulesets. A rule may include an optional priority, which is used in the conflict resolution phase of rule execution.

Forward chaining is invoked with the assert operator. The assert statement can specify a ruleset. Assert runs these rules only if the fact new; the assert! operator always runs the specified rules. Backward chaining is invoked with the find and find1 operators. These take an optional ruleset, and seek values to instantiate the following statement. Find1 resembles the Prolog cut in that it only finds the first matching value.

Additional operators particular useful in chaining include the local operator, which specifies that the following statement is not to be used as an entry for chaining, the fail statement, which causes immediate failure, the succeed statement, which always succeeds, the test operator, which succeeds only if its argument evaluates to a non-false value, the or operator, which succeeds when one of its following succeeds, and the `retry` statement, which restarts backtracking with respect to the first choice.

Functions are declared with respect to a set of named parameters. Functions, like rules, can retain their state and be called non-deterministically.

3.6.2 Implementation

ProTalk is implemented by compilation into C code that realizes an abstract machine, much like the ProLog Warren Abstract Machine. We won't go into detail on the behavior of this system, as we discuss the PrkAda abstract machine, below, except to note that: (1) The C implementation is eased by the ability in C to store pointers to actual functions. This must be faked in Ada. (2) The Ada implementation uses more flexible data structures, enabling us to ignore certain artificial limitations in the C system (e.g., the number of local variables in a function), and (3) we will be devoting greater attention to code optimization in the PrkAda version.

4. Core status

This section describes the current (alpha) core implementation. This corresponds to the substrate manager and the object manager, with a hooks for the runtime rule system. We are describing the alpha version. In that version, we attempted to by and large achieve same functionality as development ProKappa. In this section we also note those decisions we now consider mistakes. Appendix A lists the known incompatibilities between ProKappa and PrkAda. The alpha core is about 10K lines of Ada (counting semicolons; 20K, counting end-of-line characters).

4.1 Datatypes

The alpha core implements the PrkType datatype as an Ada pointer to a variant record. The enumerated type `KTYPE` represents the discriminent type of the variant; it has the following elements:

```
type KTYPE is (CLASSP,          -- class objects
               INSTANCEP,       -- instance objects
               METHODP,         -- methods
               BOOLEANP,        -- booleans
               MONINFOP,        -- monitor information
               ORIGINSLOTP,     -- origin slots—slots that contain
                                --   originally introduced material
               LOCALSLOTP,      -- local slots—slots with local
                                --    values
               INHERITEDSLOTP,  -- inherited slots—slots with
                                --   no local information
               UNIONSLOTP,      -- union slots—slot of inheritance
                                --   role union
               SYMBOLP,         -- symbols
               CONSP,           -- cons cells
               STRINGP,         -- strings
               BLANKP,          -- the unknown symbol
               ARRAYP,          -- arrays
```

```
    INTEGERP,          -- integers
    FLOATP,            -- single-precision floats
    DOUBLEFLOATP,      -- double-precision floats
    CVALUEP,           -- arbitrary Ada values, coerced to
                       --   integers
    SLOTREFERENCEP,    -- slot references: object and slot
                       --   pairs
    RAWSLOTDATAP,      -- raw slot data: a description of a
                       --   slot
    RAWFACETDATAP,     -- raw facet data: a description of a
                       --   facet
    CHARACTERP,        -- character
    EmptyArrayCellP    -- a special symbol for the unknown
                       --   array cell.
  );
```

A record of this form is a "box"; a pointer to such a record is a "ptr." Note that it is a failing of Ada that we cannot express the notion that an object is the union of the classp and instancep datatypes, a slot a union of the originslotp .. unionslotp datatypes, or that a number is the union of the interp..doublefloatp datatypes. (This could be done with a doubly discriminated record, but that would require all boxes to be doubly discriminated; this limitation seems to be corrected in the design of Ada 9X.) In retrospective: (1) It is unnecessary to allocate a special type for unknown and empty array cells. A special value would have been adequate. (2) It would, perhaps, have been better to have implemented the fundamental type as a varient record, may of whose fields would be pointers to more complex data structures. The current implementation suffers from a need to reclaim unused boxes. Making the data immediate for simpler types like integers would save much of this effort. We are following this path in the beta version.

Another problem arises from using real Ada pointers—we have inadequate control over the allocation of that storage. We plan, in the next version of this system, to employ a BIBOP (big bag of pages) implementation, where the fundamental data type is a variant record which points, when unable to hold immediate data, to a page of objects (all of which are the same kind) and to a particular object on that page. This will prove useful for garbage collection, compaction, and database activities.

4.2 Symbols

Symbols are a mapping from strings to unique data structures. Symbols are important (among other reasons) because we identify slots and facets by their symbols, and use their symbolic names to access objects. The system gains considerable efficiency through comparing symbols rather than by doing character-by-character string comparison. Achieving this efficiency requires developing a symbol table package and converting the literal strings in code to symbols. In the alpha version, this symbol table is implemented as a hash table. We used a generic package for creating cascading hash tables—hash tables that keep conflicts within a bucket on a list. When a bucket gets "too full," the cascading mechanism replaces the bucket with a cascading hash table. This is probably not as efficient, with respect to space, as rehashing the whole table but is more time efficient.

4.3 Objects

Objects in PK can be named or anonynous. Named objects have a title; anonymous objects a unique (for that application) number. All objects have a list of parents, a set of slots, a module, a properties list, and various flags that indicate properties such as whether the object is deleted, currently loading, a "system object", or an application or module. Class objects also have lists of subclasses and member children.

A critical function of an AI-style, object-system core is mapping objects and slot names to the information associated with the object's slot. This is usually done by have a single structure for each object (its slot table) and searching (with hashing) that structure for the desired slot. In the alpha version of PrkAda, we used a single global hash table, and hashed with respect to object, slot, and facet (if any). (That hash table is of the cascading kind described for the symbol table). This approach has advantages and disadvantages. In comparison to KEE, which used a fixed-size hash table per object, with list buckets off the table, it is considerably more space efficient, and not particularly less time efficient. However, it has a tendency to scatter the definition of objects through memory. This can be a disadvantage in a virtual memory system if the slot access pattern is localized with respect to the objects. However, a more serious disadvantage of this scheme is the difficult of performing certain optimizations without a well-specified object/slot data structure. That is, we would like to reduce access to compile-time constant slot names to array indices. This is hard to do with the alpha implementation. In the beta core, we are reverting to a slot array per object organization.

4.4 Slots

All slots have a data structure that points back to the parent object, and a list of facets. (The former of these is used only by the printing routines). Inherited slots (those with no local information, other than facets) point to their providing slot and object. All other slots have a set of values (or a single value). Origin slots (places in the inheritance hierarchy where a slot is introduced) keep the signature of the slot and its slotname; others point to the origin slot, and, in the case of union slots, keep their local values. (For the other inheritance roles, the combined value is the local value).

4.5 Inheritance

ProKappa is a "push" inheritance system [Filman86]. That is, when parent values change, this change is immediately propagated to all children. The algorithm works by transversing the directed acyclic inheritance link graph. The primary clevernesses of the inheritance system consist of (1) a mark phase where all elements of the subclass graph are marked, and (2) an update phase where we once again transverse the graph, updating only those nodes with no marked parents. This update process includes computing the new value of the object for the slot, unmarking the object, and recusively updating all (member and subclass) children of the object. Depending on the inheritance role, it may not be necessary to mark or update the children of a particular object. This way, it can be seen that each descendant of an object is updated exactly once.

4.6 Methods

A major incompatibility between Ada and the Lisp/KEE/ProKappa style of programming concerns functional (and procedural) objects. In Ada, functions are not first-class data types. Subprograms cannot have function-valued variables. (Ada provides the "generic" mechanism for varying behavior with respect to different procedures. Generics have the advantage of being handlable at compile time, but lax flexibility [there are some very useful things you can do with functions-as-data-values that you cannot do in plain Ada without resorting to Turing-equivalence activities such as building interpreters] and can result in great code expansion.) Our specification calls for being able to dynamically change the behavior of an object in response to a message. That is, by changing the method in a slot, an object acquires a different response to that message.

To be able to call such functions at all in Ada requires creating a mapping between the symbolic method constants and the actual Ada calls. We have taken the following approach:

(1) The system user provides a package called `methodfns`, which embodies all functions that might be called as methods or monitors.

(2) In the specification of `methodfns` is an ennumerated type `MethodFn`, which is a list of all method names in the application (whether used for messages or monitors, see below). One element of this datatype is the constant `No_Method`. For example, from CS1_Fixer application, the declaration is

```
type MethodFn is (
            No_Method,    -- in every declaration
            Prk_All_Es_Avput_method_any,
            Prk_AvputH2_method_any,
            Prk_CslResetForRules_method_any,
            Prk_IdentifyCslProblem_method_any,
            Prk_InitSymptomsAirSourceAvput_method_any,
            Prk_InitSymptomsDlAvput_method_any,
                              .
                              .
                              .);
```

The funny names (`Prk_ ... _method_any`) have been chosen for compatibility with the ProKappa ProTalk compiler, and hence, ProKappa knowledgebases.)

(3) The user supplies (accessible in the external specification, and implemented in the body of `MethodFns`), the function `Apply`. CS1_Fixer's `apply` is

```
procedure Apply (Fn : Methodfn;
                 X0,X1 : in out Ptr;
                 X2,X3,X4,X5,X6 : Ptr;
                 Ans : out Ptr) is
begin
    case Fn is
        when no_method =>
```

```
                    ans := blank;
                 when prk_All_Es_Avput_method_any =>
                    ans := prk_All_Es_Avput_method_any  (X0, X1, X2);
                 when prk_AvputH2_method_any =>
                    ans := prk_AvputH2_method_any  (X0, X1, X2);
                 when prk_CslResetForRules_method_any =>
                    ans := prk_CslResetForRules_method_any (X0, x1);
                 when prk_IdentifyCslProblem_method_any =>
                    ans := prk_IdentifyCslProblem_method_any (X0, x1);
                 when prk_InitSymptomsAirSourceAvput_method_any =>
                    ans := prk_InitSymptomsAirSourceAvput_method_any
                              (X0, X1, X2);
                 when prk_InitSymptomsD1Avput_method_any =>
                    ans := prk_InitSymptomsD1Avput_method_any
                              (X0, X1, X2);
                                      .
                                      .
                                      .
              end case;
          end Apply;
```

Immediately before the `case` statement in `apply` is a useful point for printing an collecting debugging information—all messages and monitor executions pass through this point.

(4) The specification of `MethodFns` also includes an instantiated generic (with functions renamed) for accessing the string names of methods—It allows mapping from the string "`Prk_All_Es_Avput_method_any`" to the `MethodFn Prk_All_Es_Avput_method_any`. It also provides several versions of the apply function.

The efficiency of this scheme depends on the Ada compiler. If the case statement is compiled into a dispatch from a table, then the additional overhead of a message send is roughly the cost of a slot retrieval and the cost of stacking the unused parameters and an index into the dispatch table. (This is the behavior of DEC VAX Ada). If the case statement is compiled as a series of conditional statements, then the cost of a message will be proportional to the number of different methods in the system. This can be alleviated by (1) ordering the method functions by frequency of use, and/or (2) building the apply function as a conditional tree (using the else clauses of the conditionals). Unfortunately, our experiments reveal that Verdix Ada on the SUN has this repeated conditional behavior.

4.7 Monitors

The monitor mechanism is perhaps the most complex part of the object manager. Monitors fire under five circumstances:

(1) When a monitor is attached to a slot, the attach method of a monitor fires. If certain flags are set in the monitor, the change method also fires at this time.

(2) When a value is retrieved from a slot, a WhenNeeded monitor fires

(3) BeforeChanged monitors fire when the value of a slot is modified. BeforeChanged monitors fire before the slot modification. The effects of their changes can be seen in the new slot value.

(4) AfterChanged monitors fire after slot modification. They do not effect the new slot value.

(5) Detach methods on monitors fire when a monitor is detached from a slot.

Some of the complexity arises in the current implementation because the decision to run the Attach and Detach methods is distributed to the wrong points in the code. The correct way of doing this is to modify the method inheritance role. This will be the tactic taken in the next version of the system.

4.8 Loading and saving object bases.

We anticipate moving object-bases between ProKappa and PrkAda through the ASCII object-base writer/reader mechanism. There are functions in PrkKappa for creating an ASCII representation of an object-base, and a corresponding function for reading in such a representation. For example, the following are a few lines from the ASCII representation of the CS1-Fixer application.

```
    Application Cs1  /* creating objects in application Cs1 */

    Class Cs_1  /* create a class called Cs_1 */
         /* Create a slot in Cs_1 called BreakableComponents.  This is an
            Own slot with multiple-value override inheritance.  The values
            of this slot are the objects whose names are V2h2co2, V2h2, ...
            AirSource. */
        Own MVOverride Slot BreakableComponents -> V2h2co2@, V2h2@, V1@,
  T2@,
            Rx1@, Pr1@, M1@, H2Source@, Fv2@, Fv1@, Edcm@, D1@, AirSource@
         /* Create a slot in Cs_1 called FaultyComponent.  The initial
            value of  this slot is unknown.  This slot has a facet called
            Comment with the given value. */
        Own MVOverride Slot FaultyComponent -> ?
            Facet Comment -> "If more than one value, the value is a list
                              of the possibly faulty components"
                                        .
                                        .
                                        .
         /* Create a class Orus.  Orus is a subclass of Cs_1, has own slot
            FaultyOru, and default slots (that inherit)
            ComponentFailureType, FailedOruComponent, and MemberComponents.
            */
    Class Orus
        Parent -> Cs_1
        Own MVOverride Slot FaultyOru -> ?
            Facet ValueClass -> Orus@
        Slot ComponentFailureType -> ?
            Facet Comment -> "This is the type of failure in the isolated
                              component identified in the
                              failed.oru.component slot"
            MVOverride Facet ValueClass -> Obstructed, Leaking, Drifted,
                FailedOn, FailedOff, OutOfPosition, HighCurrent
        Slot FailedOruComponent -> ?
            Facet Comment -> "This is the component identified as faulty by
                              the troubleshooting"
        MVOverride Slot MemberComponents -> ?
            Facet ValueClass -> Components@
                                        .
                                        .
                                        .
          /* Create an instance V2h2co2.  V2h2co2 is an instance of the
            classes Valves and H2EdcmOutletGroup and has values for various
            slots and facets. */
    Instance V2h2co2
        Parents -> Valves, H2EdcmOutletGroup
        Slot Value -> Normal
            Monitor Facet WhenChangedFacet -> Updateh2@
        Slot RunTestProcedure -> RunV2h2co2CycleTest
        Slot FaultState -> Ok
        Slot State -> Open
            MVOverride Facet ValueClass -> Open, HighPartialOpen,
   NormalPartialOpen, LowPartialOpen, Closed
            Monitor Facet WhenChangedFacet -> AvReverseVideo@
        Slot OperatorBreak -> ?
            MVOverride Facet ValueClass -> TemporarilyObstructed,
   Obstructed
            Monitor Facet WhenChangedFacet -> V2h2co2Mon@
        Slot OutComponent -> H2Sink
                                        .
                                        .
                                        .
         /* Instance V1DecreaseCoolingD1CorrectionProcedure has a method
            slot RunProcedure! whose value is the MethodFn
            Prk_RunV1DecreaseCoolingD1CorrectionProcedure_method_any */
    Instance V1DecreaseCoolingD1CorrectionProcedure
        Parent -> CorrectionProcedures
        Slot Status -> NotDone
        Slot `"RunProcedure!" ->
   !Prk_RunV1DecreaseCoolingD1CorrectionProcedure_method_any
                                        .
                                        .
                                        .
         /* D1Mon is a monitor; the monitor method is
            Prk_InitSymptomsD1Avput_method_any.  We could also specify at-
            tach and detach methods, priority and level for the monitor. */
    Monitor Instance D1Mon
        Parents -> ZNasaActiveValues
        Method -> !Prk_InitSymptomsD1Avput_method_any
                                        .
                                        .
                                        .
```

We have created procedures in Ada for reading and writing such object-base descriptions. The reader is a single-pass algorithm that embodies a compiled LALR (1) parse. That is, the reader can be in a number of different states, and, based on a single token look-ahead, the values of a few global variables, and the current state, progresses to the next state. As part of this state-to-state transition, the reader may take an action such as creating an object or giving a value to a slot. The only trickiness of the object-base reader is that, as a single-pass algorithm, it may encounter references to objects and slots (in the values of slots and facets) before the object or slot itself has been defined. Unfortunately, since the data structure that represents instances is different from that of classes, we can't simply create the object on first reference. We handle this problem by creating "indirect references" in slot values, remember where these indirect references are, and patching the structures after all slots have been created. This also allows us to read in several mutually-recursive object-base files simultaneously. The reader includes a separate lexical package, which is capable of reading in single box values; this lexical package is also used in the read-eval-print mechanism described below.

The PrkAda output packages have routines for printing every kind of box. The display routines have several different levels of display, including a display for the ascii reader, a detailed display, and a "pretty" display.

4.9 Debugging tools

Having both reading and writing routines, it became a simple matter to write a simple read-evaluate-print loop routine. This routine provides interactive access to all user functions in the core and assignment to local variables. For the sake of simplicity, it uses an "evaluate as you read" strategy, rather than reading the entire input and then parsing it. This has the advantage that we can tune the reading to the particular datatypes of objects required and provide immediate help on the definition of each object, but the disadvantage (?!) that erroneous input may partially evaluate.

4.9.1 Example interaction

Below is an example of the use of the system with the CS1-FIXER application. In this example, the top-level program is simply a call to the read-evaluate-print loop. Our typing is in boldface after the -> signs; comments are in braces and italics. We have reformatted a few of the lines for the narrower screen width.

```
A$run test_rep
-> 3                                            {Evaluate some constants}
 3
-> 3.141593
3.141593
-> foo                          {Unbound symbols evaluate to themselves.}
foo
-> help
_____________________________________________________________________
Use & for quote; otherwise, symbols with assigned values and function
    names evaluate; others are self-quoting.
Long_Help for full command list.
Command abbreviations are

ae                  array_elmt
afv                 add_facet_value
afvs                add_facet_values
.
.
.
```

{Here we get a complete listing of the abbreviations available in the read-eval-print mechanism. The reader is referred to Appendix B for a complete list.}

```
-> long_help
________________________________________________________________
Available commands are

add_facet_value
add_facet_values
add_list_ptr_elmt
add_value
.
.
.
```

{Once again the reader is referred to Appendix B for the complete list.}

```
-> cons(1,foo)           {foo is unbound, so it evaluates to a symbol}
( 1 . foo)
-> cons(1,cons(2,cons(3,nil)))                   {nested expressions}
( 1,  2,  3)
-> setf(zzz,cons(2,3))       {bind zzz to the result of the cons}
( 2 .  3)
-> zzz                                   {now zzz is the cons cell}
( 2 .  3)
-> &zzz                                              {quote zzz}
zzz
-> yyy                  {unbound symbols evaluate to themselves.}
yyy
-> setf(yyy,cons(yyy,zzz))                                {pun}
(yyy,  2 .  3)
-> yyy                                       {Yes, it's there}
(yyy,  2 .  3)
-> setf(xxx,cons(A,setf(xxx,  cons(12,  nil))))
                                          {setf itself evaluates}
(A,  12)
-> setf(app,  make_app("TheApp"));
        {Okay, make an application called "TheApp".  Bind app to it.}
TheApp@SystemApplication
-> setf(C,make_object(
     {Bind to C a newly created object called ClassObj.  The phrases in
                              [] are the system's parameter prompts.}

[Object]
ClassObj
,
[Module]
app,
[parents]
 nil,
[raw_slot_data]
 nil,
[instance_p]
 false);
***** Missing comma                            {Oops, a syntax error.}
Exception: Missing closing parenthesis              , location:  86.
-> C
       {But in this strange environment, it's after the facet, so C has
```

```
            been bound.}
ClassObj@
-> make_slot(                                 {Make a slot on ClassObj}
[Object]
C,
[Slot Name]
"TheSlot",
[Raw Slot Data]
nil,
System exception.
-> find_slot(C,TheSlot);
    {Look for it under the wrong name.  The system interprets this as a
                                            plain symbol and a list.}
find_slot
(C, TheSlot)
-> is_slot(C,TheSlot)                             {Try the right name.}
FALSE
-> make_slot(C,TheSlot,mrsd((1,2,xxx,yyy),MVOver,
                  {Looks like the syntax error killed the last make_slot.
                                                           Try again.}
[facets/()]
,
))
-> get_values(C,TheSlot);
  {Inside a list, the symbols were not evaluated.  We need to have them
                                                    at the top level.}
( 1,  2, xxx, yyy)
-> ms(                   {Make another slot, S, using the value of xxx.}
[Object]
C,S,mrsd(cons(3,xxx),   SelfFirstUnion,   nil))
-> display(C)                                  {Show the object in C}
Class ClassObj@

   SelfFirstUnion Slot S ->  3, A,  12
   MVOverride Slot TheSlot ->  1,  2, xxx, yyy
-> mo(Child,  TheApp,              {Make another object, get an error.}
[parents]
(C),
[raw_slot_data]
nil,
[instance_p]
true);
Exception: Child Of nonclass                          , location:  50.
-> mo(Child,  TheApp,  (ClassObj@),  nil,  true)          {Do it right}
Child@
-> ddisp(Child@)                    {Show the inheritance in Child@}
Child@
   Module: TheApp@SystemApplication
   Parents: (ClassObj@)
   Object kind: ORDINARY_OBJ
   Is deleted?: FALSE
   Is system?: FALSE
   Properties: ()

   Default SelfFirstUnion Slot S [UNIONSLOTP from: ClassObj@] ->  3, A,
     12
   Default MVOverride Slot TheSlot [INHERITEDSLOTP from: ClassObj@]
     ->  1,  2, xxx, yyy
-> add_value(Child@,S,  22)              {Add 22 to Child@'s S slot.}
-> avs(Child@,  TheSlot,  (A,  B,  Child@))
              {Add two symbols and an object to Child@'s TheSlot slot.}
-> ddisp(Child@)                                          {Show Child@}
Child@
   Module: TheApp@SystemApplication
   Parents: (ClassObj@)
   Object kind: ORDINARY_OBJ
   Is deleted?: FALSE
   Is system?: FALSE
   Properties: ()

   Default SelfFirstUnion Slot S [UNIONSLOTP from: ClassObj@]
       ->  22,  3,  A, 12
   Default MVOverride Slot TheSlot [LOCALSLOTP from: ClassObj@]
       -> A, B, Child@
-> load("Csl.txa")              {Load the Csl_Fixer knowledge base}
Csl@SystemApplication
-> ddisp(V2h2co2@)  {Show one of the objects in that knowledge base.}
V2h2co2@
   Module: Csl@SystemApplication
   Parents: (Valves@, H2EdcmOutletGroup@)
   Object kind: ORDINARY_OBJ
   Is deleted?: FALSE
   Is system?: FALSE
   Properties: ()

   Default Override Slot ComponentFailureType [INHERITEDSLOTP
       from: Orus@] -> ?
   Default Override Slot ControlConnection [INHERITEDSLOTP from:
       Components@] -> ?
   Default Override Slot FailedOruComponent [INHERITEDSLOTP from:
       Orus@] -> ?
   Default Override Slot FaultState [LOCALSLOTP from: FaultStateClass@]
       -> Ok
   Default Override Slot InComponent [INHERITEDSLOTP from: Components@]
       -> ?
   Default Override Slot Line [INHERITEDSLOTP from: Components@] -> ?
   Default MVOverride Slot MemberComponents [INHERITEDSLOTP from:
       Orus@] -> ?
   Default Override Slot OperatorBreak [INHERITEDSLOTP from:
       ControlComponents@] -> ?
           MVOverride Facet ValueClass -> TemporarilyObstructed,
               Obstructed
           Monitor Facet WhenChangedFacet -> V2h2co2Mon@
   Default Override Slot OutComponent [LOCALSLOTP from: Components@] ->
       H2Sink
   Default MVOverride Slot RunTestProcedure [LOCALSLOTP from:
       Components@] -> RunV2h2co2CycleTest
   Default Override Slot State [LOCALSLOTP from: ControlComponents@] ->
       Open
           MVOverride Facet ValueClass -> Open, HighPartialOpen,
                                          NormalPartialOpen,
                                          LowPartialOpen,
                                          Closed
           Monitor Facet WhenChangedFacet -> AvReverseVideo@
   Default Override Slot Value [LOCALSLOTP from: Components@] -> Normal
           Monitor Facet WhenChangedFacet -> Updateh2@
```

```
-> disp(Diagnose@)                                     {Show another}
Class Diagnose@

   Own Method Slot Identify! -> !PRK_IDENTIFYCS1PROBLEM_METHOD_ANY
   Own Method Slot Reset! -> !PRK_CS1RESETFORRULES_METHOD_ANY
           Facet Comment -> "Reset the kb for fault diagnosis ie., it
                             puts unknown in all unit slots set by
                             fault diagnosis rules"
           Facet ValueClass -> Method
   Own Method Slot Show! -> !PRK_SHOWBREAKABLES_METHOD_ANY
-> send(Diagnose@,    Reset!)
                               {Try it out!  Send diagnose a reset message.}

System reset to normal operating conditions
()
-> set_value(V2h2co2@,OperatorBreak,    Obstructed);
                                    {Break something--say, obstruct V2h2co2.}

Fault entered.
->  send(Diagnose@,    Identify!)
    {Try to diagnose the problem.  What follows is the program output.}

Faulty component detection report
Suspected faulty component(s)

-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --

    Component V2h2co2@ in ORU Fca@ failed Obstructed
    Component Pr1@ in ORU Fca@ failed Obstructed
    H2 subsystem has problem: LowH2Flow()
-> ddisp(V2h2co2@)                              {Display V2h2co2 again.}
V2h2co2@
   Module: Cs1@SystemApplication
   Parents: (Valves@, H2EdcmOutletGroup@)
   Object kind: ORDINARY_OBJ
   Is deleted?: FALSE
   Is system?: FALSE
   Properties: ()

   Default Override Slot ComponentFailureType [LOCALSLOTP from: Orus@]
       -> Obstructed
   Default Override Slot ControlConnection [INHERITEDSLOTP from:
       Components@] -> ?
   Default Override Slot FailedOruComponent [INHERITEDSLOTP from:
       Orus@] -> ?
   Default Override Slot FaultState [LOCALSLOTP from: FaultStateClass@]
       -> UncorrectedFault
   Default Override Slot InComponent [INHERITEDSLOTP from: Components@]
       -> ?
   Default Override Slot Line [INHERITEDSLOTP from: Components@] -> ?
   Default MVOverride Slot MemberComponents [INHERITEDSLOTP from:
       Orus@] -> ?
   Default Override Slot OperatorBreak [LOCALSLOTP from:
       ControlComponents@] -> Obstructed
           MVOverride Facet ValueClass -> TemporarilyObstructed,
                                          Obstructed
           Monitor Facet WhenChangedFacet -> V2h2co2Mon@
   Default Override Slot OutComponent [LOCALSLOTP from: Components@]
```

```
            -> H2Sink
        Default MVOverride Slot RunTestProcedure [LOCALSLOTP from:
            Components@] -> RunV2h2co2CycleTest
        Default Override Slot State [LOCALSLOTP from: ControlComponents@]
            -> Open
                MVOverride Facet ValueClass -> Open, HighPartialOpen,
                                               NormalPartialOpen,
                                               LowPartialOpen,
                                               Closed
                Monitor Facet WhenChangedFacet -> AvReverseVideo@
        Default Override Slot Value [LOCALSLOTP from: Components@] -> Normal
                Monitor Facet WhenChangedFacet -> Updateh2@
    -> exit                                                         {Bye.}
```

5. Compiler status

We are developing a compiler for the logic programming language ProTalk. The compiler is designed to produce executable Ada programs. The compiler knows little about primitive operations and data structures, but rather concentrates on general issues of environment and control. Instead of having a specialized knowledge about a large number of constructs, the compiler handles only a small set of operations which reflect the semantics of the lambda calculus. It achieves this simplification by using an intermediate language which is closely related to the LISP dialect SCHEME.

Existing approaches to compiling logic programming languages are based on the Warren Abstract Machine, which directly supports high-level logic programming operations. Building these operations into the base machine makes it difficult to implement a wide variety of program optimizations. In addition, the use of idiosyncratic program semantics has isolated logic programming compiler technology from the main stream of compiler research. In contrast, we have developed a generic approach to compiling high level logic programming constructs within a framework which has already been applied to a variety of programming languages.

The PrkAda system is designed to deliver a complete ProKappa application in an Ada environment. As PrkAda compiler is not incremental, it is not an appropriate tool for the development of ProTalk programs. The commercial ProTalk compiler and C-based ProKappa development environment serve that purpose. By having the complete application available, the PrkAda compiler is able to perform global analysis and optimization of ProTalk rules and functions. Later we show how global optimization techniques to support efficient application delivery in the PrkAda run-time environment.

In the following sections we will discuss the most important features of ProTalk from the perspective of compiler design. We will describe the program analysis and code generation techniques used in the PrkAda compiler and present the compiler architecture, briefly describing the purpose of each pass. Finally we will conclude with our plans for future work.

5.1 Programming Language Issues

From a compiler design perspective, the most interesting features of ProTalk are its support for logic variables, nondeterministic backtracking, and pattern-directed invocation. We will show how the PrkAda compiler handles these features as well briefly discuss some the alternatives we considered. There is brief description of ProTalk in this report; the reader is referred to the ProTalk Reference Manual for detailed information about the ProTalk language, its syntax, and semantics.

All compilers replace programming constructs in a source language with equivalent constructs in a target language, choosing those target constructs which provide the greatest efficiency. All compilation at some point exercises the primitive operations of the target, "underlying machine." For conventional compilers that compile to machine code, "integer addition" and "floating point multiplication" are examples of such operators. The ProTalk compiler treats the PrkAda core routines as part of its target machine model. Hence, the compiler freely uses the routines, such as list processing and slot storage, provided by the core.

Another difference between ProTalk and Ada is their respective typing disciplines. Ada is a very strongly typed language whereas ProKappa is very weakly typed and relies heavily on run-time type checking. Again, the PrkAda run-time core handles this difference between the languages by supporting run-time type checking with a universal data type called "box" which encapsulates the raw ProKappa data with a corresponding type tag.

In general, neither data types (such as lists or objects) or ordinary control logic pose any problems for the PrkAda compiler. All data structures issues are handled by the PrkAda run-time core and any conventional control logic can be easily translated into Ada. However, there is no direct support in the PrkAda target language for the language abstractions of backtracking, pattern-directed invocation, and logic variables. The implementation of backtracking in Ada is further complicated by the fact that Ada does not support function pointers, unlike languages such as C or Lisp.

5.1.1 Pattern-directed invocation

The ProTalk language includes both a backward and forward chaining rule facility. The actual chaining of one rule into another is essentially a function call mediated through a pattern matching mechanism called unification. However, whereas an ordinary function will have a unique set of parameters, a rule may have multiple sets of parameters. This is because each entry point in a rule defines a potentially distinct set of parameters. For example, consider the following ProTalk rule:

```
bcrule DummyRule in DummyRules
{
 if:
   ?comp.Value == High;
   ?comp.Sensor == ?sen;
   ?sen.Trend == High;
 then:
   ?comp.ComponentFailureType +== ReadsHigh;
   ?sen.Status = Normal;
}
```

The two statements in the "then" clause are both potential entry points to DummyRule. If the first statement is matched then ?comp is the sole parameter of the rule. If the second statement is matched then ?sen is the parameter of the rule.

It is in the first phase of the compiler that we make the parameters of a rule explicit. We accomplish this by doing a global analysis of how the rules of an application match up. Having done so, we generate a ProTalk function corresponding to the entry point that is used. The newly created function is computationally equivalent to the original rule. The only difference is that it has a unique set of parameters. Note that since the PrkAda is designed to work in a batch mode, there is no need to generate code for entry points that are not used in an application.

5.1.2 Logic Variables

All variables in ProTalk are what we call logic variables. The term is borrowed from the logic programming community. (In fact, ProTalk variables are restricted relative to true logic variables such as Prolog). A reference to a logic variable is like a reference to an ordinary variable with one important difference. When a logic variable is unbound and is used in a predicate it becomes bound when the predicate is computed, assuming that the predicate otherwise is true. For example, given the following ProTalk predicate statement:

```
?comp.Sensor == ?sen;
```

If ?sen is not bound when the statement is executed, then it acquires the value of the expression ?comp.Sensor. In an conventional language (such as Ada) it is never ambiguous as to which variables are the parameters of a program.

5.1.3 Nondeterminism

Nondeterminism is a technique for implementing search. We call an expression nondeterministic when it selects its return value from a set of possible values, using some arbitrary selection criteria. The idea is that it is the responsibility of ensuing computations to determine if the value generated is "correct". If it isn't, then control is returned to the generating expression and another value is selected.

Like most serial logic programming languages, ProTalk uses a depth-first approach to non-deterministic search. Each ProTalk statement in ProTalk can be likened to a node in a search tree. The execution of a ProTalk program traverses this tree in a depth-first manner. Whenever it is necessary to make a nondeterministic choice, a "choice point" is created which captures the current state of the computation, along with sufficient information to generate the "next" value in the value set. Program control proceeds down the search tree until it is determined that the most recent choice was incorrect. In that case control is returned to the most recent choice point and the "next" value is selected. The choice point is updated an control proceeds down the tree again. If a choice point has no more values then it is discarded and the preceding choice point is visited. This process of returning to the previous choice point is called "backtracking".

Here is an example of backtracking in ProTalk taken from the ProTalk reference manual. Here we seek an affordable `Foil`. Suppose that the class `Foil` has three instances: `FL1` whose `Price` is `1000`, `FL2` whose `Price` is `200`, and `FL3` whose `Price` is `30`. We now consider what happens when the following ProTalk code is executed:

```
find ?x = direct instanceof Foil;              /* S1 */
Print(?x, "is a foil");                        /* S2 */
find ?x.Price < 100;                           /* S3 */
Print(?x, "is an affordable Foil");            /* S4 */
```

A choice point is created at S1 when it is first encountered. The execution of S1 will select a value and bind the variable `?x` to it. In this example, `?x` will be bound successively to the values `FL1`, `FL2`, and `FL3`. After each binding of `?x` in S1, statement S2 will be executed, followed by S3. If the test in S3 fails, as it does in the first two cases, then control is returned to S1. When the test of S3 succeeds the third time, control proceeds to S4.

5.2 Compiler Architecture

The PrkAda ProTalk compiler performs a series of transformations on an source program which result in an equivalent but simpler program. Each transformed program is simpler then its predecessor in one of two measures. Either it a uses a smaller set of instructions or it is expressed in terms of instructions which are more explicit (i.e. more primitive).

The transformations used by the ProTalk compiler are:

(1) Parsing and conversion to AST (abstract syntax tree).
(2) Preliminary ProTalk analysis.
(3) Conversion to PIL (PrkAda Intermediate Language).
(4) Intermediate analysis.
(5) CPS conversion.
(6) Environment analysis
(7) Code generation.

In (1), a textual representation of a ProTalk program is parsed and converted into a set of objects which form an abstract syntax tree. Each object has additional slots which are used by ensuing phases of the compiler for caching information. During (2), a preliminary analysis is performed of the binding of logic variables and the parameters of ProTalk functions and rules are made explicit. Phase (3) converts ProTalk abstract syntax trees into equivalent intermediate program trees. The intermediate forms are then analyzed for side effects and scoping of ordinary variables (4). After slots have been populated by intermediate analysis, a program is converted in (5) to CPS (Continuation Passing Style) form. As a result, all control flow (including backtracking) is made explicit. In (6), additional analysis of variable references results in the assignment of slot values which support the code generation for environments. This results in making variable environment processing explicit. Phase (7) takes the analyzed CPS program and converts it into executable Ada.

The PrkAda compiler concentrates on issues of environment and control constructs. Low-level issues concerning data structure manipulation are left to be handled by the Ada compiler. For example, an Ada compiler must have specific knowledge about data structures such as numbers, arrays, and strings. The PrkAda compiler knows little about such things, so we just pass them along as Ada output code. We leave it the Ada compiler to determine how to implement primitive data structures.

5.2.1 Parsing and Abstract Syntax Trees

The PrkAda compiler transforms a textual representation of a ProTalk program into an abstract syntax tree (AST) in which each node of the tree is an object. This is in contrast to using an ad hoc representation of a program. Because each node of an AST is an object, we can apply a generic set of routines to access and manipulate these structures. Our ProTalk parser is specified by a augmented BNF grammar. The parser is automatically generated from the grammar by a YACC-like facility.

As the parser operates on a program, it generates objects according to the rules of the grammar. Each grammar rule, with its associated object definition, specifies the types of objects to be constructed when the rule is applied. The following is a simple example from the ProTalk parser:

```
PT-PLUS  ::= [ lexpr "+" rexpr ] builds PT-PLUS,
```

When this rule is applied it constructs an object of the type PT-PLUS and assigns values to the slots "lexpr" and "rexpr". The object definition for PT-PLUS has type definitions for the slots "lexpr" and "rexpr." Slot type definitions act as constraints on the values which can be assigned to the respective slots.

5.2.2 Preliminary ProTalk Analysis

The primary function of this phase of the compiler is to disambiguate the binding status of every ProTalk variable reference and to make explicit the parameters for every ProTalk function. The ProTalk features of pattern-directed invocation and logic variables make it ambiguous whether a given variable reference is bound or unbound at compile time. There are three questions of interest with respect to a variable reference. The first is whether it is an initial reference. The second question is whether the variable reference is a parameter reference. The parameters are the specified inputs of a ProTalk rule or function. The third question of interest is whether a variable is bound or unbound. The answer to the third question is related to the first two.

Only initial references to variables may be unbound. If an initial variable reference is not a parameter reference is must be unbound. If an initial variable reference is a parameter reference then its binding status cannot be inferred at compile time without additional information. By doing a global analysis of the call graph of the ProTalk rules and functions of an application, we are able to propagate binding status information. As a result, we are able to minimize the amount of run-time checking of variable binding status. By generating code for each initial parameter reference, the PrkAda compiler is able to assign a unambiguous binding status of either bound or unbound to all other logical variable references.

5.2.3 Conversion to Intermediate Language

A set of rewrite rules translate ProTalk programs into equivalent PIL programs. The translation of ProTalk to PIL currently requires four passes. Conversion is non-trivial because the compiler must perform control flow analysis to identify backtrack program entry points and variable scopings.

Knowledge of variable scopings enables the next compiler phase to analyze a program for side effects and to make all variable references explicit. The identification of backtrack entry points are necessary for the CPS conversion phase, in which the compiler introduces explicit control constructs.

The language we have chosen (called PIL) is a small subset of Scheme, a simple yet powerful dialect of Lisp. Recent work on integrating logic programming into Scheme has produced a simple, yet elegant formulation of backtracking in Scheme. We have used that formulation in PIL. The result is a language which is compact, capable of expressing the program control necessary for backtracking, and whose clean semantics lends itself to program analysis and optimization. The PrkAda compiler translates nondeterministic ProTalk programs into equivalent PIL programs.

Here are the eight PIL primitive expression constructs and their meanings:

```
exp ::= constant-exp | var-exp | lambda-exp | letrec-exp |
    if-exp | if-exp | set!-exp | call-exp
constant-exp ::= object
var-exp ::= identifier
lambda-exp ::= [formals body]
letrec-exp ::= [binding-list body]
if-exp ::= [expl exp2 body]
set!-exp ::= [identifier exp]
call-exp ::= [head arguments]
ccs-exp ::= call-arg
```

The following are auxiliary definitions.

```
call-arg ::= lambda-exp
body ::= exp
formals ::= identifier*
binding-list ::= [(identifier exp)*]
head ::= exp
arguments ::= exp*
```

`var-exp:` An identifier reference.
`constant-exp:` A constant. Most languages allow numeric and string constants.
`lambda-exp:` A procedure.
`if-exp:` An IF statement. <expl> must evaluate to a boolean. If True <exp2> is evaluated, else <exp3> is evaluated.
`letrec-exp:` This is used to define mutually recursive procedures.
`call-exp:` A procedure application. The head and arguments are evaluated, then the evaluated head is applied to the evaluated arguments.
`set-exp:` An assignment statement. The <exp> is evaluated then the assignment takes place.

5.2.4 Intermediate Analysis

Intermediate analysis is composed of two distinct phases. The first is called variable analysis. Its purpose is to make explicit the semantics of variable references and to determine which variables are destructively changed, (i.e. side-effected). Every programming language has implicit evaluation rules about how a variable reference should be interpreted. For example, an ordinary assignment statement has a l-value and a r-value. The l-value is interpreted as the location of the variable. The r-value is interpreted as the new contents of the location. Intermediate analysis makes variable locations explicit, makes assigning values to locations explicit, and makes the accessing of a variable location explicit. This simplifies code generation by avoiding the need to repeatedly re-examine the context of a variable reference. Once this analysis is performed, the meaning of a variable reference never changes. The analysis of variable references simplifies the task of determining which locations can be side-effected. Knowledge of side-effects is necessary to determine if certain types of code motion are permissible during optimization.

The other component of intermediate analysis is called code linearization. The purpose of linearization is to transform a program into an equivalent program in which the order of evaluation of all expressions is fixed and explicit and all intermediate values are named (placed in temporary variables).

5.2.5 CPS conversion

Standard CPS conversion takes a program with procedure calls and returns and returns an equivalent program with in which procedure calls never return. In CPS form, each function call passes an additional parameter, a function called the continuation. The continuation represents the next computation to perform. Instead of returning, each function passes its result to its continuation. As a simple example of CPS conversion, consider the append function in Scheme.

```
(lambda (x y)
  (letrec
   ((append-fun
     (lambda (x y)
       (if (null x)
           y
           (cons (car x) (append-fun (cdr x) y))))))
   (append-fun x y)))
```

The long (PIL) form of this function is:

```
(lambda (x y)
  (letrec
   ((append-fun
     (lambda (x y)
       (if (call null x)
           y
           (call (lambda (t1)
               (call (lambda (t2)
                  (call (lambda (t3)
                        (call cons t1 t3))
                      (call append-fun t2 y)))
                (call cdr x)))
          (call car x))))))
   (call append-fun x y)))
```

The standard CPS converted form of this is:

```
(lambda (x y cont1)
  (letrec
   ((append-fun
     (lambda (x y cont2)
       (call null x
        (lambda (t4)
          (if t4
            (call cont2 y)
            (call car x
              (lambda (t1)
              (call cdr x
              (lambda (t2)
              (call append-fun t2 y
              (lambda (t3)
              (call cons t1 t3 cont2))))))))))))))
   (call append-fun x y cont1)))
```

The following is a specification of standard CPS conversion algorithm in terms of rewrite rules:

```
(cps-convert-lambda (lambda (<arg>*) <body>))
   ==>
`(lambda (,<arg>* , <new-cont>)
   ,(cps-convert <body> <new-cont>))

(cps-convert <var id> <cont>)
   ==>
`(call ,<cont> ,<var id>)

(cps-convert <constant k> <cont>)
   ==>
`(call ,<cont> ,<constant k>)

(cps-convert (letrec ((<id-1> <lambda-1>)
                      ...
                       (<id-n> <lambda-n>))
          <body>)
   <cont>)
   ==>
`(letrec ((<id-1> ,(cps-convert-lambda <lambda-1>))
      ...
     (<id-n> ,(cps-convert-lambda <lambda-n>)))
   ,(cps-convert <body> <cont>))

(cps-convert (call <var id> <arg>*) <cont>)
   ==>
`(call ,<var id> ,<arg>* ,<cont>)

(cps-convert (call (lambda (<arg>) <body) <exp>) <cont>)
   ==>
```

```
`(call ,<cont> ,(cps-convert-lambda <lambda-exp>))
```

We needed to extend standard CPS conversion in order to support backtracking. Whereas a deterministic program in standard CPS form has only one continuation, a nondeterministic program in our approach always has two continuations. One continuation handles what to do next on success (i.e the return) and the other continuation represents what to do if backtracking is required.

The implementation of this extended CPS conversion is straightforward. The modified Cps-Convert takes three arguments instead of two, since we now have two continuations. After CPS conversion, every user procedure will get two additional continuation arguments. In addition, we extended PIL with one more expression type called call-ccs [Ruf89, Ruf91]. Call-ccs takes a single argument, a lambda procedure, and calls it with the current continuations.

5.2.6 Environment Analysis

Environment analysis supports the generation of optimized code for ProTalk environments during run-time. Its principle goal is to identify those variables can be safely stored on the stack and which variables must be allocated on the heap. Heap-allocated variable environments are required for backtracking. When backtracking occurs, a continuation is executed which may need to restore the current environment to an earlier computational state. We cannot rely on accessing the environment on the stack because the function which defined the environment may have been exited, which would result in it being popped off the stack.

The problem of identifying those variable which need to be stored on the heap is equivalent to the problem of identifying those closures (i.e. lambda expressions) which are being treated as data. When a closure is passed as data we say it must be fully closed (i.e it's environment must be allocated on the heap). In CPS-form, all variables are defined in closures. If it can be determined that a given closure expression can only occur in the function position of any reachable function calls then we the closure is said to be open. This means that the closures variables can be safely allocated on the stack. However, if the closure could potentially be used in a non-function position of any reachable function call which exits the surrounding lambda, then the closure must be closed and its associated variables need to be heap allocated.

During environment analysis, all closures are closed until proven open. Closure analysis is complicated by the fact that a closure is not truly open if it lexically occurs in another, fully closed closure. It is necessary to determine, for each node in the abstract syntax tree, the set of variables referred to within closed functions at or below that node.

5.2.7 Code Generation

Code generation is the last phase of the compiler and has received the least attention. After phase six (environment analysis), all significant issues concerning environment and control have been resolved. This should make it relatively straightforward to translate an analyzed PIL program, in CPS form, to Ada.

5.3 Comparison to other Work

Conventional compiler technology does not address the issue of how to support backtracking. There are many papers on techniques for compiling Prolog, a language which, like ProTalk, uses depth-first backtracking to support nondeterministic search [Warren83, Komatsu86]. There appears to be little connection between recent Prolog compiler research and mainstream compiler technology. This is unfortunate since mainstream compiler research has amasses a large body of techniques for optimizing code. A major reason for this is most Prolog compilers use an idiosyncratic model called the Warren Abstract Machine (WAM). It is used both as a intermediate representation and as a target language.

The WAM has the virtue of being a easy target language for compilation since its instruction set directly supports the high-level operations of depth-first backtracking and pattern matching. However, the WAM does not provide the simple instructions which are necessary for an optimizing compiler. It also has a overly concrete model which makes it difficult to add "simpler" instructions. Given sufficient knowledge, an optimizing compiler should be able to substitute high-level instructions with lower-level and more efficient ones. As a result, WAM-based compilers for Prolog are severely hampered in their ability to perform code optimization. WAM is also deficient as an intermediate language since its machine model is not conducive for the types of code analysis necessary for significant code optimization.

Instead of adopting the WAM model, we chose another approach, based on recent trends in compiler research [Kelsey89, Weise89]. It is centered around the use of a simple intermediate language based on the lambda calculus. The guiding principles of this approach are uniformity, simplicity, and maximal explicitness. Uniformity is achieved by expressing all constructs in common terms. Simplicity is achieved by using a minimal intermediate language. Maximal explicitness requires that constructs should be as precise as possible, allowing the compiler can reason about and optimize them.

We have extended this approach by augmenting CPS conversion to support languages with failure continuations. As a result, we have established a correspondence between the optimization of backtracking and the optimization of closures. Making backtracking explicit has made logic programs amenable to optimization techniques used by an emerging compiler technology which is independent of any specific programming paradigm.

6. Using PrkAda with ProKappa

While it is possible to develop an application in ProKappa/Saber solely with the core and ProTalk, or to develop an application using only the delivery libraries (the PrkAda core; much as conventional development is done), we believe that it will prove convenient to be able to develop applications in the ProKappa/Saber C-based environment with Ada-language methods (thereby getting the best of both worlds). Such methods should then transfer transparently to the delivery environment. The goal of this segment of the work was to enable writing methods in Ada and running them in ProKappa/Saber. This section describes the mechanisms involved. Keep in mind that these comments apply to the Verdix Ada compiler and the ProKappa system, both running on Sun-4/Sparc stations.

While the Ada compiler and GNU are compatible in certain respects (use of the stack and linker format), they have several major incompatibilities. The major difficulties we encountered were:

1. The Ada environment is set up for an Ada top-level. Certain things get done "automatically" by the Ada system, especially the before-the-main-program initialization routines for Ada packages and procedures.

2. The Ada system makes different use of the registers than GNU C. In particular, Ada keeps the stack limit in general register G4; GNU C assumes that this register is available for the computations of any procedure. Thus, typical GNU-C compiled code trashes the value in this register.

3. Ada data representations are different than those of C. Two important differences are:

   3a. Strings. Both systems encode strings as packed arrays of characters, but Ada keeps a count (just before the string) of the number of characters in the string, while C strings are null terminated.

3b. ProTypes. ProKappa has its own variant record type, the ProType, which encompasses the useful datatypes of ProKappa (e.g., symbols, objects, integers, and floats). The encoding of this datatype is somewhat non-standard: the three lowest bits are used to encode the datatype tag. A ProType which is a pointer thus uses a 29-bit pointer and a three bit tag; ProType integers use a quarter of the tag space, and thus have a 30-bit range and appear to be multiplied by 4. Of course, Ada pointers are 32 bits, and Ada does not support arithmetic operations on pointers (except through unchecked conversions).

6.1 Our example:

We base the following discussion around the following example. ProKappa methods are functions of at least two arguments. These functions are placed as the values of slots. When one sends a message, "`Mess`," to an object, "`Obj`," the system retrieves the value of the "`Mess`" slot of "`Obj`." This should be a function, which is applied to "`Obj`," "`Mess`," and the other arguments of the message. We wish to write a method, "`mymethod`," which is sent a (`PrkType`) number, "`val`". This code instructs an object that has it as a method to look up the value of its own "`friend`" slot, which contains another object. This object then places one more than "`val`" in the "`score`" slot of its friend. (Of course, the domain semantics of this example are nonexistent; the imporant thing is that we demonstrate a variety of ProKappa behavior within Ada.)

```
with prk; use prk;

-- A vanilla method. When sent this message, with a number val,
-- the recipient finds a friend (in his friend slot), and stores
-- in the friend's score slot one more than the given value.
function mymethod (self : ptr; slotname : ptr; val : ptr)
        return integer is

   friendsym : ptr;
```

```
        friend     : ptr;
        i          : integer;

    begin
        friendsym := MakeSymbol ("friend");
        friend    := GetValue (self, friendsym);

        i         := unbox (val) + 1;

        SetValue (friend, MakeSymbol ("score"), box(i));

        return i;
    end mymethod;
```

Keep in mind that a "`ptr`" is ProType. This method is calling the `MakeSymbol`, `GetValue`, and `SetValue` functions of the ProKappa core. The functions `unbox` and `box` are used to convert PrkTypes to and from native types. For expository purposes, the code is a bit more verbose than it needs to be; the entire function could be a single line without declarations.

6.2 Overall architecture

Recalling the problem with register G4, noted above, we need to ensure that whenever we enter Ada code this register is set properly. Thus, making our example work requires four pieces of code, as illustrated in Figure 3.

1. The Ada code for mymethod, shown above.
2. The Ada Prk package, which provides an Ada view of ProKappa functions and datatypes, such as ptr, MakeSymbol, and SetValue.
3. A C interface between the ProKappa core and mymethod.
4. A C interface between the Prk package and the ProKappa core.

6.2.1 The Prk Package:

The Prk package is Ada code that defines the `ptr` datatype (as a 32 bit, uninterpreted number), its subtypes, and provides an interface to each ProKappa user function. It needs to be hand generated, but, except for string conversions, the coding seems is relatively automatic. Most of the package is simply specifies that these functions are in C (pragma interface) and providing the name for the C function to call (`pragma interface_name`, in code part 4). `MakeSymbol` from a string requires a little more work, as it must convert the Ada representation of a string to the C representation.

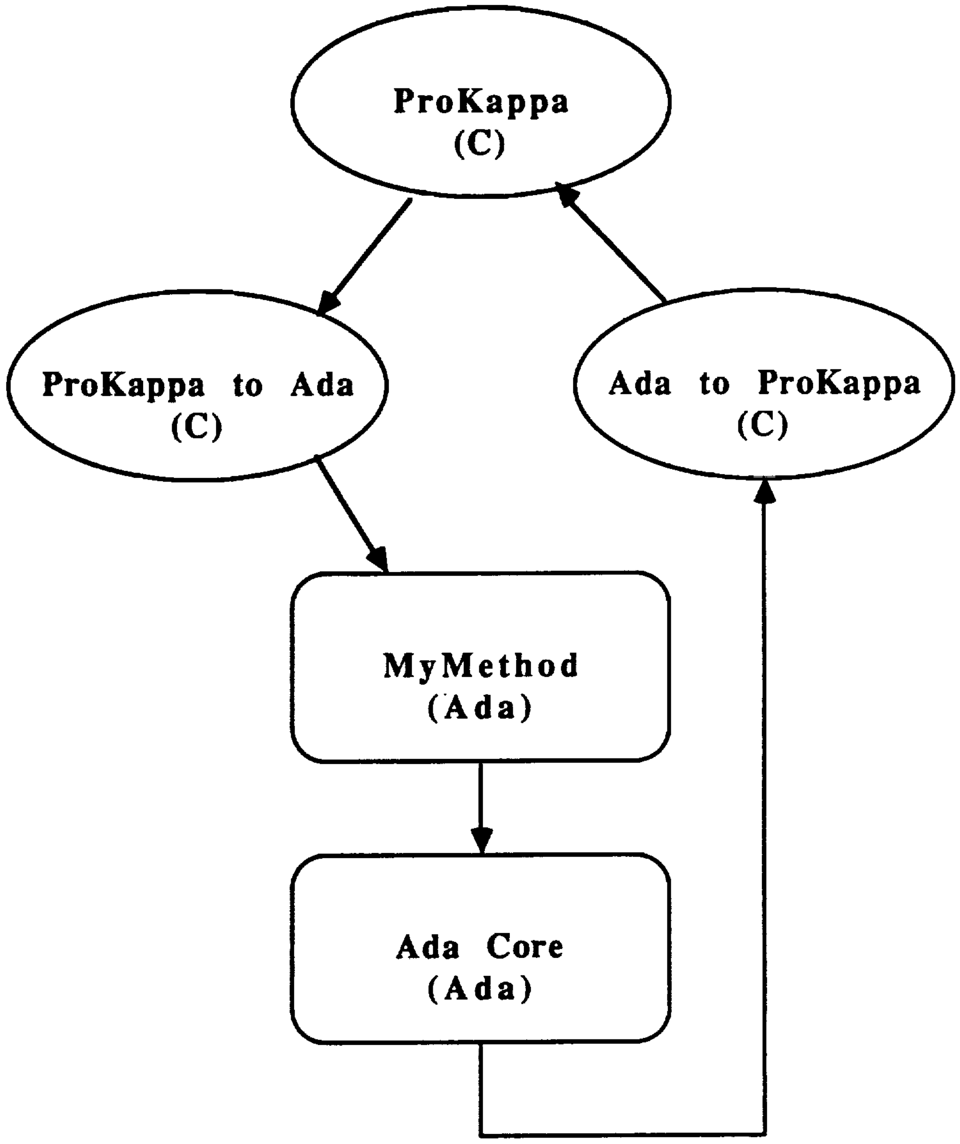

Figure 3. The Method Integration Architecture.

```
with system; use system;

package prk is
    -- to be extended or modified as the type KTYPE varies:
   type KTYPE is (CLASSP, INSTANCEP, ORIGINSLOTP, LOCALSLOTP,
        INHERITEDSLOTP, UNIONSLOTP, SYMBOLP, CONSP, STRINGP,
        BLANKP, ARRAYP, INTEGERP);

   type c_string is access STRING (1..1000);

   type ptr is range -(2**31) .. (2**31)-1;


   subtype OBJECT is PTR;      -- (CLASSP .. INSTANCEP).
   subtype SLOT is PTR;              -- (ORIGINSLOTP .. UNIONSLOTP)
   subtype ORIGINSLOT is PTR; -- (ORIGINSLOTP);
   subtype SYMBOL is PTR;     -- (SYMBOLP);
   subtype LIST is PTR;       -- (CONSP);
   subtype KSTRING is PTR;    -- (STRINGP);
   subtype KARRAY is PTR;     -- (ARRAYP);
   subtype KINTEGER is PTR;   -- (INTEGERP);

   NIL :  PTR ;          -- the empty list, none.
   NUL :  PTR ;          -- nothing; blank

   function makesymbol   (x : string) return ptr;
   function c_makesymbol (x : address) return ptr;
      pragma interface      (c, C_makesymbol);
      pragma interface_name (c_makesymbol, "_APrkMakeSymbol");

   function findobject (x : ptr) return ptr;
      pragma interface      (c, findobject);
      pragma interface_name (findobject, "_APrkFindObject");

   function findobject (x : string) return ptr;

   function getvalue (u : ptr; s : ptr) return ptr;
      pragma interface      (c, getvalue);
      pragma interface_name (getvalue, "_APrkGetValue");

   procedure setvalue (u : ptr; s : ptr; v : ptr);
      pragma interface      (c, setvalue);
      pragma interface_name (setvalue, "_APrkSetValue");

   function unbox_to_i (x : ptr) return integer;
      pragma interface      (c, unbox_to_i);
      pragma interface_name (unbox_to_i, "_AUnboxToInt");
   function unbox (x : ptr) return integer renames unbox_to_i;

   function box_to_i (x : integer) return ptr;
      pragma interface      (c, box_to_i);
      pragma interface_name (box_to_i, "_ABoxToInt");
   function box (x : integer) return ptr renames box_to_i;

   procedure print (x : address);
      pragma interface      (c, print);
      pragma interface_name (print, "_APrint");
end prk;

package body prk is
    t : string (1..1000);  -- this doesn't seem to work when t is
                           -- local to the procedure makesymbol.

    function makesymbol (x : string) return ptr is
    begin
       t(1..x'last) := x;
       t(x'last+1)  := ascii.nul;
       return c_makesymbol(t'address);
    end makesymbol;

    function findobject (x : string) return ptr is
    begin
```

```
            return findobject (makesymbol (x));
        end findobject;
    end prk;
```

6.2.2 The interface functions:

A most important detail of the interface to Ada is the need to set general register G4. This is accomplished with the following C code.

```
    #include "g4.h"

    int G4val;

    void SetG4 ()
    {
        asm("sethi %hi(_G4val), %g1");
        asm("ld [%g1+%lo(_G4val)],%g4");
    }

    int APrkInit ()
    {
        asm("sethi %hi(_G4val), %g1");
        asm("st %sp, [%g1+%lo(_G4val)]");

        G4val = G4val - 25000;

        SetG4();
        return G4val;
    }
```

Note we must call the function `APrkInit` at least once before calling any Ada code. This stores the value of the end of stack in G4val. (We assume the stack is 25000 elements long.) We call SetG4 when we call an Ada function or return from a call back into Ada.

Appendix C lists the Happy C interface to the ProKappa core. We basically provide a Happy C function for each core function.

The following code presents mymethod to the ProKappa/C environment. Note that it does a little type conversion itself. A function of this form is required for each Ada-language method. However, it is straightforward to write a program to generate mechanically such functions from the Ada program text.

```
    PrkType mymethod (PrkType self,
                      PrkType slotname,
                      PrkType val)
    {
       int ans;

       SetG4();
       ans = A_mymethod(self, slotname, val);
       return (PrkType) ans;
    }
```

Detail on using these files with the Verdix Ada system is also described in Appendix C.

One interesting attribute of this arrangement is that we now have a dynamically linking Ada system. We can revise the Ada function mymethod, recompile it, reload the new object file into the existing ProKappa/Saber environment and continue working.

6.3 Limitations of this approach:

The following cautions should be noted about the results of this experiment.

1. Problem: This stuff will likely work only for simple Ada methods—top level functions that are not part of a package, don't do any "before main" initialization, and don't rely on exceptions.

   Resolution: Write only simple methods. While it is likely that it would be possible to call the "before main"initialization Ada code and packages, it is unrealistic to expect the Ada exception mechanism to work in the ProKappa environment.

2. Problem: One must be very careful about types. In particular, ProKappa likes to use the bottom three bits of a number for tags; Ada has no knowledge of this convention.

   Resolution: Create an interface function in Happy C for each Ada method. Create an interface function in C foreach ProKappa function.

3. Problem: This only works for Verdix Ada on the Sun 4/Sparc machine.

   Resolution: After all, the ProKappa development environment only works on Suns and HP's. Presumably, similarmechanisms can be developed for other compilers and instantiations of ProKappa.

Bibliography

[Bachant89] Bachant, J., and Soloway, E., "The Engineering of XCON," *Communications of the ACM, vol. 32*, no. 3, 1989, pp. 310--319

[Brownston85] Brownston, L., Farrell, R., Kant, E., and Martin, N, *Programming Expert Systems in OPS5*, Reading, Massachusetts: Addison-Wesley, 1985.

[Clocksin84] Clocksin, W. F., and Mellish, C. S., *Programming in Prolog*, New York: Springer-Verlag, 1984.

[Fikes85] Fikes, R. E. and Kehler, T., "The role of frame-based representation in reasoning," *Communications of the ACM, vol. 28*, no. 9, 1985, pp. 904--920.

[Filman84] Filman, R. E. and Friedman, D. P., *Coordinated Computing: Tools and Techniques for Distributed Software*, New York: McGraw-Hill, 1984.

[Filman86] Filman, R. E., "Retrofitting objects," ACM Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA-87), Orlando, Florida, 1987, pp. 342--353.

[Forgy82] Forgy, C. L., "Rete: A Fast Algorithm for the Many Pattern/Many Object Matching Problem," *Artificial Intelligence 19*, 1982, pp. 17--37.

[Hewitt71] Hewitt, C., "Description and Theoretical Analysis (Using Schemata) of PLANNER: A Language for Proving Theorems and Manipulating Models in a Robot," MIT AI Laboratory TR--258, 1971.

[Kelsey89] Kelsey, R. and Hudak, P., "Realistic Compilation by Program Transformation," in *Proc. ACM Symposium on the Principles of Programming Languages*, Austin, Texas, Jan. 1989, pp. 281-292.

[Komatsu86] Komatsu, H., Tamura, N., Asakawa, Y., and Kurosawa, T., "An Optimizing Prolog Compiler," in E. Wada (Ed.), *Logic Programming '86: Proceedings of the 5th Conference*, Springer Verlag, Berlin, pp. 104-115.

[Ruf89] Ruf, E., and Weise, D., "Nondeterminism and Unification in LogScheme: Integrating Logic and Functional Programming", *Proc. of the 4th ACM conference on Functional Programming*, pp. 327-339, Imperial College, London, September, 1989

[Ruf91] Ruf, E., and Weise, D., "Using Types to Avoid Redundant Specialization," to appear in *Proc. ACM Sigplan Symposium on Partial Evaluation and Semantics Based Program Manipulation*, June 1991.

[Warren77] Warren, D. H. D., "Implementing Prolog - Compiling Predicate Logic Programs," University of Edinburgh D.A.I. Research Report 39--40, May, 1977.

[Warren83] Warren, D. H. D., "An Abstract Prolog Instruction Set," Technical report number 309, Artificial Intelligence Center, SRI International, 1983.

[Weise89] Weise, D., "Advanced Compiling Techniques," Lecture notes, Computer Science Department, Stanford University, 1989.

Appendix A. Limitations and restrictions of the current system

We have attempted, in the development of the PrkAda system, to be as compatible as possible with the ProKappa C environment. (This may, in fact, be a mistake; such compatibility restricts the possible optimizations) The development of the PrkAda core has so far revealed a number of restrictions on use and differences in behavior with respect to the ProKappa core. It is worthwhile mentioning these.

1. Garbage collection. ProKappa has an automatic garbage collector. In the alpha version of PrkAda, explicit "free" calls must be make to deallocate storage, both cells and boxes. The system is also likely to be less

than perfect about freeing all the storage it itself uses. (Such mistakes are bugs.) Improving this situation is a major part of the development of the next version of the system.

2. Graphics. ProKappa has extensive facilities for developer C and end-user graphics. We provide no graphic operations or primitives for interrogating the X event stream. (Our code does not even need to be run under X.)

3. Exceptions. The ProKappa signal/exception mechanism has not been implemented. Instead, keeping in the spirit of Ada, exceptions are signaled with the Ada raise statement and handled by Ada exception handlers. That is, one can't make a slot value be an exception. (It would be relatively straightforard to extend the box datatype to include an exception enumeration, and to provide a module, similar to the one used for messages, to raise the appropriate exceptions. However, we believe this has little value.) We have created an eclectic collection of exceptions, reflecting the particular implementation of the PrkAda core, and the lack of specification of the ProKappa error collection.

4. Stack arrays and lists. The purpose of a stack array is to automatically reclaim storage on exit from a routine. Ada doesn't allow pointers to objects on the stack. Hence, we cannot directly implement stack arrays in Ada. In the Beta version, we expect to implement some form of data pools for similar effect. That is, conses and array can be designated to come from a particular pool; there will be operations to create pools, make a pool the current pool, and free all the storage of a pool.

5. Functions as explicit objects. Ada doesn't have functions as explicit objects. Instead, in PrkAda, we manipulate a user-defined enumerated type whose names correspond to the user functions, as described above.

6. Application variables/environment variables. Applications can have variables associated with them, much like Unix variables, and one can also interrogate the environment for its variables. Since we are building portable Ada code, it seems wiser to allow the user to call specific operating system variable mechanisms in his or her own code. The effect of application variables that are not meant to inherit from the Unix environment can be better achieved by using slots on particular "environment" objects in the application.

7. Loops. ProKappa has C macros Loop, LoopObjectSlots, and LoopSlotsFacets to loop through the elements of a list, through the slots of an object, and through the facets of a slot, respectively. Their syntax is somewhat awkward. Ada doesn't have macros. Instead, we implemented functionality of these macros with a set of generic functions. We have expanded this set to include other looping constructs, such as the ability to loop through multiple lists simultaneously.

8. The date and time datatypes. These are a planned extension for the next version of ProKappa. We plan to extend our system to include them too.

9. Slot descriptors. Slot descriptors are an internal ProKappa datatype; it was a mistake to mention them in the user manuals. We have not replicated this mistake.

10. Datatypes. We have some slight differences in the set of datatypes allowed for slot values. We exclude certain ProKappa types (e.g. errors, as exceptions aren't objects in Ada, and slot descriptors) and include a few others (e.g., different implementation varieties of objects and slots, particular constants.) However, all the major datatypes are represented.

11. Loaded applications. As a runtime system, we do not include the concept of an application not being loaded. We interpret "loaded" for applications as the same as "not deleted."

12. Type strings. The PrkTypeName function has been changed to use strings appropriate for our datatype, and to map boxes to strings, not integers. This is another function that really shouldn't be documented at the user level.

13. Copying strings. The ProKappa MakeString function "boxes" a string. It has an optional argument that determines if the string is copied. PrkAda's Make_String function always copies its argument, as Ada doesn't permit pointers to constants or elements on the stack.

14. Returned values. Various ProKappa functions and procedures take arguments that are pointers to values. The pointed-to values are then set by the function. For example, PrkGetValueOrValues has an "address of a PrkBool" argument, where it sets the underlying (pointed to) boolean to indicate if the slot is single or multi-valued. In general, this technique is used in C to get the effect of out parameters. On one hand, Ada does not allow functions that have out parameters. On the other hand, Ada procedures really do have out parameters. We often resolve this conflict by providing two overloaded forms of such routines, one a procedure where both the functional answer and the additional information are out parameters, and the other a function that returns only the functional information.

15. Message parameters. Parameters to messages must be boxes. Messages have a maximum number of parameters (currently 5, easily raised).

16. Boxed booleans vs. booleans. Many ProKappa routines take booleans as arguments or return booleans as results. While there is a boxed boolean datatype, this is not always the most convenient form for such arguments. That is, IF wants a real boolean. Should the PrkAda routines deal in boxed booleans or plain booleans? The resolution of this conflict lies in part in providing overloaded versions of these routines. However, it is then to simple to create ambiguous overload resolution paths. We are currently exploring providing different named versions of functions, depending on whether the desired answer is a boolean or boxed boolean. In general, it appears that most uses of functions that take or return booleans are more convenient with ordinary booleans; the unusually-named functions will thus be used for the boxed versions.

17. Monitor firing order. Due to implementation stupidity, PrkAda is somewhat inconsistent about the order of firing of attachment and detachment monitors. In general, if the order of equivalent events with respect to the monitor system is not specified. That is, if attachment of a monitor at the class level causes the attach method of that monitor to fire in several children, the order of that firing is not specified in ProKappa and may differ between ProKappa and PrkAda.

18. Static facets. PrkAda, alpha version, does not have the notion of a static facet.

19. Saving applications. The only saving/loading mechanism in the alpha version of PrkAda is the ASCII application loader/saver.

20. Unimplemented functions. Certain functions, such as the ability to rename slots, have not be implemented. These functions are appropriate for a development environment, where the developer may need to make changes in the overall representations. However, we view these functions to be inconsistent with the notion of a delivery environment; their existence precludes many useful optimizations.

Appendix B. Read-eval-print help

A complete listing of the abbreviations and functions available in the read-eval-print mechanism.

```
-> help
```

```
Use & for quote; otherwise, symbols with assigned values and function
    names evaluate; others are self-quoting.
Long_Help for full command list.
Command abbreviations are

ae                  array_elmt
afv                 add_facet_value
afvs                add_facet_values
append              append_lists
av                  add_value
avs                 add_values
car                 list_first
cdr                 cdr
children            object_children
cons                make_cons
copy                copy_list
ddisp               detailed_display
disp                display
fo                  find_object
getprop             get_property
gfv                 get_facet_value
gfvl                get_facet_values_list
gfvs                get_facet_values
gv                  get_value
gvs                 get_values
gvsl                get_values_list
last                list_last_cons
```

```
length              list_length
list                make_list
load                load_ascii_app
member              find_list_elmt
mf                  make_facet
mm                  make_monitor
mo                  make_object
mrfd                make_raw_facet_data
mrsd                make_raw_slot_data
ms                  make_slot
nconc               destructive_append_lists
nth                 list_nth
ows                 objects_with_slot
parents             object_parents
remprop             remove_property
rfv                 remove_facet_value
rfvs                remove_facet_values
rplaca              set_list_first
rplacd              set_list_rest
rv                  remove_value
rvs                 remove_values
sae                 set_array_elmt
send                send_msg
setf                assign
setprop             set_property
setq                assign
sfv                 set_facet_value
sfvs                set_facet_values
sop                 set_object_parents
sv                  set_value
svs                 set_values
```

```
-> long_help
```

```
Available commands are

add_facet_value
add_facet_values
add_list_ptr_elmt
add_value
add_values
app_classes
app_instances
app_modules
app_name
append_lists
apps
array_elmt
array_fill_count
array_size
array_to_list
assign
attach_monitor
cdr
clear_monitor_flags
clear_slot_flags
collection_length
copy_list
```

```
delete_app
delete_facet
delete_list_elmt
delete_list_ptr_elmt
delete_module
delete_object
delete_slot
destructive_append_lists
detach_monitor
detailed_display
display
exit
fill_array
find_app
find_list_elmt
find_module
find_object
find_symbol
free_list
free_objects_with_slot_list
free_type
get_c_value
get_facet_inheritance
get_facet_value
get_facet_value_or_values
get_facet_values
get_facet_values_list
get_local_facet_value_or_values
get_local_value_or_values
get_method_fn
get_msg_fn
get_property
get_slot_type
get_type
get_value
get_value_or_values
get_values
get_values_list
help
is_ancestor_object
is_anonymous_object
is_app
is_array
is_array_equal
is_c_value
is_char
is_collection
is_cons
is_deleted_app
is_deleted_module
is_deleted_object
is_double_float
is_empty_list
is_equal
is_facet
is_fixnum
is_float
is_instance
```

```
is_list
is_list_equal
is_loaded_app
is_loaded_module
is_loaded_object
is_method
is_module
is_monitor
is_multi_value_facet_inheritance
is_multi_value_slot_inheritance
is_number
is_object
is_raw_facet_data
is_raw_slot_data
is_single_float
is_slot
is_slot_reference
is_static_facet
is_string
is_symbol
is_system_object
list_first
list_last_cons
list_length
list_nth
list_rest
list_to_array
load_app
load_ascii_app
load_module
long_help
make_app
make_array
make_c_value
make_cons
make_double_float
make_facet
make_list
make_method
make_module
make_monitor
make_object
make_raw_facet_data
make_raw_slot_data
make_single_float
make_slot
make_slot_reference
make_stack_list
make_string
make_symbol
method_fn_name
module_classes
module_instances
module_name
mon_info_filter
mon_info_flags
mon_info_is_multi_value_slot
mon_info_monitor
```

```
mon_info_object
mon_info_priority
mon_info_slot_name
monitor_level
monitor_priority
move_object
noop
object_app
object_children
object_module
object_name
object_parents
objects_with_slot
print
put
put_line
remove_facet_value
remove_facet_values
remove_property
remove_value
remove_values
rename_app
rename_facet
rename_module
rename_object
rename_slot
reset_system_object
save_app
save_ascii_app
save_module
send_msg
set_array_elmt
set_array_fill_count
set_facet_inheritance
set_facet_value
set_facet_values
set_list_first
set_list_rest
set_monitor_flags
set_monitor_level
set_monitor_priority
set_object_parents
set_property
set_slot_flags
set_slot_inheritance
set_slot_type
set_system_object
set_value
set_values
slot_inheritance
slot_origin
slot_reference_object
slot_reference_slot_name
switch_app
test_monitor_flags
test_slot_flags
verbose_print
```

Appendix C. Code for running Ada in ProKappa

C.1 The Happy C interface to the ProKappa core

The following functions are the Happy C interface to the ProKappa core. Note that we rely on C and the Happy C compiler for some of the type conversions, and that we always call SetG4 before returning to Ada. In a complete system, this code would need to be generated, semi-mechanically, for all user functions in ProKappa.

```
#include "g4.h"

PrkSymbol APrkMakeSymbol (char* s)
{
   PrkSymbol sym;
   sym = PrkMakeSymbol (s);
   SetG4();     -- Note, we have to call SetG4 just before
        -- returning.  Hence, the use of the temporary in
        -- these functions.
   return sym;
}

PrkType APrkFindObject (PrkType s)
{
   PrkType val;
   val = PrkFindObject (s);
   SetG4();
   return val;
}

PrkType APrkGetValue (PrkType o, PrkType s)
{
   PrkType val;
   val = PrkGetValue (o, s);
   SetG4();
   return val;
}

void APrkSetValue (PrkType o, PrkType s, PrkType v)
{
    PrkSetValue (o,s,v);
    SetG4();
    return;
}

int AUnboxToInt (PrkType x)
{
    SetG4();
    return (int) x; -- note the cast.
}

PrkType ABoxToInt (int x)
{
   return (PrkType) x; -- note the cast.
}

void APrintf (char* x)
{
    printf ("<%.20s>, [%x]\n", x, x);
}
```

C.2 The Verdix Ada system

Writing programs with the Verdix Ada system requires the following steps:

1. Establishing an Ada library, using a Verdix program for library creation.

2. Compiling Ada functions with respect to that library, using the Verdix compiler.

3. Linking those programs with respect to a designated "main" procedure, using the Verdix linker, which (1) creates an initialization program and the data for that program, (2) creates a command file for the standard Unix linker that links elements of the Verdix library, the initialization program, and the users code, and (3) invoking this command file to create an executable image.

Using the -V switch on the Verdix Ada loader, we get a listing of which files are required to build a particular executable. In general, we notice the need to load (using the Saber load command) the following object files from the Ada library:

```
/vads6_0/standard/.objects/system01
/vads6_0/standard/.objects/system02
/vads6_0/standard/.objects/v_i_types01
/vads6_0/standard/.objects/v_i_types02
/vads6_0/standard/.objects/link_block01
/vads6_0/standard/.objects/link_block_b01
/vads6_0/standard/.objects/library.a
```

Additionally, we need the object files for the prk package and the object files for the particular methods we've written. Here, prktop is a pseudo-top-level that provides certain essential constants.

```
.objects/prktop.o
.objects/toprk01
.objects/toprk02
.objects/toprktop01
.objects/mymethod01
```

We can now install the C function mymethod on a slot and send messages.

Appendix D. Contractual notes

D.1 Personnel

The following people worked on this contract in the reporting quarter:

Robert E. Filman, Senior Scientist, Principal Investigator.
Roy Feldman, Scientist.

D.2 Travel

Dr. Filman and Mr. Feldman attended the National Conference on Artificial Intelligence in Anahiem in July.

D.3 Publications

None this quarter

D.4 Estimated progress

At the end of this quarter, the work on the contract is 49% complete. We plan on completing an additional 16% of the contract in the coming quarter, and on completing substancially all the work on the contract by the end of February, 1992. These plans may be effected by the availability of additional funding or changes in personnel.

Legal Notice

IN-61-CR

84336

P.62

NASA

Report Documentation Page

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
4. Title and Subtitle Compiling Knowledge-Based Systems to Ada Annual Report		5. Report Date August 1991
		6. Performing Organization Code
7. Author(s) Robert E. Filman Roy D. Feldman		8. Performing Organization Report No. PrkAda-I
		10. Work Unit No.
9. Performing Organization Name and Address IntelliCorp, Inc. 1975 El Camino Real West Mountain View, California 94040		11. Contract or Grant No. NAS 8 - 38488
12. Sponsoring Agency Name and Address Marshall Space Flight Center Huntsville, Alabama 35812		13. Type of Report and Period Covered Annual; 7/90-6/91
		14. Sponsoring Agency Code
15. Supplementary Notes		
16. Abstract		
17. Key Words (Suggested by Author(s)) ProKappa, Ada, Expert systems, Knowledge-based systems, Compilation Automatic programming	18. Distribution Statement	
19. Security Classif. (of this report) Unclassified	20. Security Classif. (of this page) Unclassified	21. No. of pages / 22. Price

NASA FORM 1626 OCT 86