Compiling Knowledge-Based Systems from KEE to ADA

Robert E. Filman
Conrad Bock
Roy Feldman

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

January 11, 1990

Abstract

The dominant technology for *developing* AI applications is to work in a multi-mechanism, integrated, knowledge-based system (KBS) development environment (e.g., KEE^{TM}). Unfortunately, systems developed in such environments are inappropriate for delivering many applications — most importantly, they carry the baggage of the entire Lisp environment and are not written in "conventional" languages. One resolution of this problem would be to "compile" applications from complex environments to conventional languages. Here we report on our first efforts to develop a system for compiling KBS developed in KEE to ADA^{TM}. We call this system KATYDID, for KEE/Ada Translation Yields Development Into Delivery. KATYDID includes early prototypes of a run-time KEE core (object-structure) library module for ADA, and translation mechanisms for knowledge structures, rules, and Lisp code to ADA. Using these tools, we have (not quite automatically) compiled part of a simple expert system to run in a purely ADA environment. This experience has given us various insights on ADA as an artificial intelligence programming language, potential solutions of some of the engineering difficulties encountered in early work, and inspiration on future system development.

1 Overview

Our long-term goal is that after a KEE prototype has "settled down," one pushes a few buttons and the system produces several files of ADA subprograms (and data files for these programs) that encode the application. After transferring this code to the target machine, it is compiled and linked with a library of KEE support subprograms

and with the application or interface in which it is to be embedded. (Thus, in ADA terms, the product of the translation process is an ADA library subprogram that uses an ADA-language, KEE-core library subprogram.) This optimum environment would include facilities for examining and modifying the code produced by the translation, for further development of the application in ADA, and for incorporating these changes in the KEE environment for repeated translation.

Here we report on our first efforts to develop a system for compiling KBS developed in KEE to ADA. We call this system KATYDID, for Kee/Ada Translation Yields Development Into Delivery. KATYDID has components that run in the KEE and REFINE^{TM} Lisp-based environments and in arbitrary ADA environments. The components of the KATYDID system are:

- **The KATYDID core.** The core is an ADA-language library which supports the underlying KEE-like object-system base functionality.
- **The knowledge-base dumper.** This system, written in KEE and REFINE, generates structures that can be used by the KATYDID core to recreate a version of an application knowledge base.
- **The rule compiler KRICKIT** (Kee/ada Rule Invocation Compiler KIT). KRICKIT is a REFINE program that takes a set of backward chaining KEE rules and a set of pattern-directed queries on those rules. It compiles them into ADA functions that, when used in conjunction with the KATYDID core, have the same effect as running those queries in KEE.
- **The Lisp-To-ADA translator CIKADA** (Commonlisp In Kee to ADA). CIKADA is a REFINE program that translates the Lisp code of a KEE application to ADA.

As part of our early efforts, we have succeeded in compiling part of a NASA prototype system called CS-1/FIXER [8]. CS-1/FIXER, by reasoning from a structural model, diagnoses and suggests repair actions for a space-based air purification system. Our demonstration system lacks the graphics and display panel mechanisms of the original; we have also hit some severe size limitations of our ADA compiler. However, using our typescript, menu-based interface, we can break the system components and obtain diagnoses of these breakages from their symptoms using the CS-1/FIXER rules. Additionally, we are able to dynamically construct object systems from the keyboard or through ADA programs.

Article overview. This article contains four more sections. In Section 2, we discuss the run-time KATYDID core. In Section 3, we turn to mechanisms for translation from Lisp-based to ADA-based programs, discussing the compilation process itself, compiling Lisp code to ADA, and compiling rules. In Section 4 we discuss our observations on the KBS compilation process and in Section 5, present our plans for extending this work.

2 The KATYDID core

The KEE system is an environment for building knowledge-based systems. KEE is object-centered and integrates several AI problem-solving paradigms, including (1) frames, (2) inheritance, (3) access-oriented programming (demons), (4) object-oriented programming, (5) multiple-worlds, (6) truth maintenance, and (7) production rules, with (8) facilities for querying, altering, and displaying the resulting structures. The first six of these form the KEE core. Unfortunately, space limitations preclude our giving a detailed description of KEE. Additional detail can be found in several of the references [2,5,6].

The KATYDID core is an approximately 10,000 line ADA program that implements the functionality of much of the KEE core. (In certain interesting respects, its functionality exceeds that of KEE.) The gross structure of the KATYDID core is perhaps best illustrated as having three layers: (1) *Structural operations*: The basic types of the KATYDID core and the primitive operations on these types; (2) *Core functionality*: The code that implements the "KEE-like" behavior of the KATYDID core. This includes the functions that store and retrieve slot values, create units and slots, invoke active values, index slots for the rule system, and perform inheritance; and (3) *Core interface*: The presentation of the functionality of the core in the conventional KEE form to applications. That is, the functions of the user's manual are at this level.

Most of the application code rests on the user level. However, ADA's lack of dynamic method binding requires that application methods and demons be inserted (by the compiler) under the "core functionality" layer.

Primitive types and data structures. In a strongly-typed language such as ADA, there is benefit to not multiplying types. Towards this end, in KATYDID we implemented a uniform treatment of knowledge bases, units, slots, facets and so forth. Thus, as in Opus [3,4], KATYDID implements a fully recursive unit structure. The data type layer supports primitives for accessing and modifying the fields of an object.

As an (effectively) embedded system, the KATYDID core needs a language for communicating with its callers. It is not enough just to provide objects, since much of the access to KEE objects is symbolic. This leads us to our second datatype, the *symbol*. Like Lisp, KATYDID has the equivalent of oblists and symbol properties.

ADA is strongly typed. If objects are to have values, ADA must know their type. We selected six types as appropriate for slot values: integers, floats, booleans, objects, symbols, and lists. We defined a *box* as a variant record of two fields, one of which indicates the type of the content of the box and the second, the actual data. Lists are *box_lists*; the *car* of such a list is itself a box, the *cdr*, a box_list. In essence, this duplicates the "tagged values" that are invisibly supported by Lisp systems. In KATYDID we have to carry the tagging along with each box and explicitly unwrap the box to get to its data.

The datatype layer supports a moderate range of functions for manipulating box_lists — functions for placing a value into a box, taking it out, comparing boxes and values for equality, consing values onto box_lists, `car`, `cdr`, `rplaca`, and `rplacd`, and a small complement of list manipulation primitives such as membership and set union.

Core functionality. The KATYDID core implements the following important classes of

KEE functionality: object creation; retrieval from symbolic name to data structure; the specification of inheritance links, including the retrieval of the closure of the inheritance link relationships; inheritance, including several predefined inheritance roles; indexing slots of a specified names for the rule system; object properties; and the storing and retrieving of values of slots, complete with coercion between data types and active values. We comment on some of the more interesting aspects of these mechanisms below.

- **Inheritance links and shared structures.** Like KEE, KATYDID supports two varieties of links, superclass/subclass and class/instance. One can make any object be of subclass (subset) or instance (set element) of any other. Such a declaration establishes a *base link* between the two. A link between two objects, parent and child, causes every member sub-object (e.g., member slot) of the parent to exist in the child. If the link is a subclass link, the child has a member sub-object; if the link is an instance link, the child has an own sub-object. If the child already has a sub-object of that name, the two sub-objects are merged. If not, the child shares the sub-object with its parent. We implement this by creating a *derived link* between a non-shared sub-object and its parent sub-object. This allows the normal inheritance process to work between them. The process is repeated recursively down the sub-object hierarchy.

  The greatest complexity of the object manipulation comes through the manipulation of objects that share substructures. More precisely, if a parent, G, has a member slot, S, and G's descendant C does not have any local information about that slot, C shares most of the data structure of S. This saves considerable storage at the expense of dramatically increasing program complexity. When a shared object acquires some local information (such as a value on one of its slots or facets or multiple parents that do not share the same structure) it becomes necessary to "make that object local" to its superior. This may involve search, as the process needs to split the children formerly noted on the parent between the parent and the newly-local child.

- **Inheritance of values.** Slots acquire values both from the explicit, local placement of that value on the slot (*local values*) and because the slot is a descendant of other (member) slots which have values. The slot's visible values are based on combining these through an inheritance role. In KATYDID we implemented several of KEE's inheritance roles: *union*, which does a set union of all parent values with the child's local values; *unique* (or *variable values*), which ignores all parent values; and *override values*, which uses local values, if any, otherwise the values of *some* parent. We extended these roles with a *prototype* role which acquires its parent's values at object creation or linkage but then acts like the role unique.

- **Slot values.** KATYDID includes the full range of functions of storing into and retrieving from the values of slots. The core supports two kinds of active values, AVGETs and AVPUTs, but not the active values associated with attaching and detaching from a slot.

User layer. KATYDID presents in the user layer most of the significant functionality of the KEE core described above. The semantics of KEE often specify that a particular argument can be either a symbol or an object. When such a function has a single such parameter, this leads to the user layer providing overloaded versions of such functions, for symbol, object, and boxed values.

Knowledge-base construction. Rather than develop a symbolic representation of a knowledge base, we chose to have the knowledge-base dumper create ADA procedures that, by calling functions such as `create_unit` and `put_values`, realize the appropriate structures. This is accomplished by constructing Lisp functions to re-create the knowledge base and then running those functions through the Lisp to ADA translator discussed below. This "compiled KB" mechanism has the advantages that (1) knowledge base loading is extremely quick, (2) the code itself can be modified, and (3) it was straightforward to implement. This approach has the disadvantages that (1) the code for creating the KB remains part of the running core image (2) there is no natural mechanism for saving changed knowledge bases, and (3) the size of the files produced can cause problems for some compilers.

Omissions. The KATYDID core, as currently implemented, lacks some of the KEE core functionalities. Most critical of these are (1) the lack of functions for eliminating structure, i.e., deleting objects or removing inheritance links, (2) mechanisms for symbolically loading and saving knowledge bases, (3) valueclass and cardinality checks, (4) obscure inheritance roles, and (5) the ATMS and KEEworlds.

3 System translation

In this section, we (1) overview the nature of the translation process and then discuss how we apply translation to problems of (2) compiling Lisp code and (3) compiling KEE rules.

KATYDID Translation. The purpose of the KATYDID system is to take a KEE application and deliver it in ADA. Almost all KEE applications include some Lisp programs; many of them include KEE rules. The KATYDID approach to rules and programs is to *compile* them into ADA.[1] We call the KATYDID rule compiler KRICKIT (KEEADA Rule Invocation Compiler KIT) and the Lisp-To-ADA translator CIKADA (Commonlisp In KEE to ADA).

Instead of writing our compiler in a procedural language like C or Pascal, we have chosen to use a *transformational implementation* [1]. A program transformation system takes other programs as input and manipulates them by iteratively applying rules. Typically, a rule replaces a program structure in the source language with one in the target language. It is also common that individual rules are *correctness-preserving*: the new program structure computes the same results as the one it replaced.

[1] A contrasting approach would be to build, in ADA, both Lisp and rule interpreters. However, since the delivery of KEE applications in ADA is the primary goal of this project, we chose to translate rules and Lisp code directly into ADA. This approach enables us to produce significantly more efficient systems.

Because writing a compiler is a relatively complex task, a number of tools have been built to support compiler development. These tools are often called *compiler-compilers*, even though they usually deal only with part of the problem of compiler construction. In this work, we have used the Refine system [7,9] as our compiler-compiler. Refine is characterized by a grammar facility for expressing language syntax and a tree structure of knowledge objects to represent a program's evolving semantics. In KATYDID we use four different Refine grammars, one for each of ADA, Lisp, the KEE rulesystem, and an intermediate language in KRICKIT. Given a grammar, REFINE performs as both a scanner and a parser generator. Whenever an expression is parsed, it is done with respect to a specific grammar.

Translating Lisp into Ada. The most critical problem in translating Lisp into Ada is that of typing. Lisp is weakly typed—any variable can be dynamically bound to any object. The system tags or remembers the type of all data. Ada is strongly typed and requires types to be specified at compilation. We identified three approaches to the type problem: type inference, overloading, and explicit declaration.

Type inference. Type inference is the inference of the type of Lisp variables from information implicit in the program. In the example below, a variable is set to the result of a function. The return type of the function can be deduced and used as the type of the variable `ch`.

```
(defun test1 (j)
  (let (ch)
    ...
    (setq ch (integer-to-character j))
    ...
  ))

(defun integer-to-character (i)
    (cond
      ((= i 1) 'a')
      ((= i 2) 'b')
      ((= i 3) 'c')
      (t       'd')))
```

The complete transformation cannot be made until the type of `j` is determined and we know whether `test1` is a function or a procedure.

The above example shows two characteristics of type inference. The first is that some inferences are not formally correct. In the example, we are deducing the type of `ch` from the type of the value to which it is set. This could be incorrect, as another assignment statement to the same variable may use a different type. It is often the case that we need to make such *heuristic* decisions about type inference. We can make our inference formal by proving that the variable is not set to another type, or by restricting our use of this deduction to the code that occurs between one setting of the variable and another.

Similarly, we might use heuristic inference to deduce the type of `i` in the function `integer-to-character`. In that case, all comparisons of `i` are to integers, so we can conclude that in normal programming practice `i` will be an integer. However, in Lisp = may be redefined by the user, so this deduction cannot be made formal without further analysis.

A second characteristic of type inference is that it can make use of the flow of data through a program. This is shown in the deduction of the return type `integer-to-character`. We can deduce that the result will be in the set {`'a'`, `'b'`, `'c'`, `'d'`}. All these are of type `character`, so we know the result type is `character`. We call this *data flow* type inference. A critical component of data flow inference is identifying the return points of a subprogram and the values returned at those points.

Data flow type inference can be heuristic. For example, if `integer-to-character` lacked its "otherwise" clause (`'d'`), we might still infer that the result type is `character`, even though formally it could include return the `list` or `symbol nil`.

Overloading. The second approach to handling the difference in type strength between Lisp and Ada is to use Ada's overloading—having multiple definitions of a single subprogram, discriminated on the basis of their argument and result types. This pushes the type discrimination problem off onto the ADA compiler.

There are three main difficulties involved with relying on overloading

1. Many versions of the same function are needed to cover all the possible cases of argument types. With ten arguments of seven types we must write (on the ADA side) 7^{10} ($\approx 2.8 \times 10^8$) versions of the same function. (Such a function was contemplated for the KATYDID core.) Few ADA compilers can handle this.

2. It is possible, with overloading, to create ambiguous programs that fail to compile.

3. Overloading cannot completely replace type inference in Lisp. It cannot, for example, determine the return type of function `test1`.

Thus overloading can only be an element, combined with dataflow analysis, in the resolution of the type inference problem.

Explicit declarations. The third approach to type differences is simply to rely on the user to make Lisp type declarations. Many CommonLisp type declarations can be translated directly into Ada declarations with only a minimal amount of type inference. While *formally* reliable, this approach is both tedious and prone to *human* error.

Rule Compilation. The KATYDID system supports KEE rules with KRICKIT, the backward chaining rule compiler. KRICKIT takes KEE rules and a set of pattern-directed queries as inputs. Its output is a set of semantically equivalent Ada functions. KRICKIT has four components:

- **Rule parameterization.** Rule parameterization is the process of determining the "directionality" or "flow" of the variables in a rule. A given rule can be parameterized in different ways, based on its use in different contexts. Each parameterization results in a separate function.

- **Intermediate language.** The syntax of rules is significantly different than that of a procedural language. KRICKIT translates the rules into an intermediate language, so as to replace the implicit enumeration of rule languages with iterative and enumerative constructs of conventional languages.

- **Ada code generation.** From the intermediate rule language, KRICKET translates to ADA code, using processes similar to the translation mechanisms described in the last section. A relatively small number of ADA constructs are used in the resulting code.

- **Rule Compiler Runtime.** The only additional facility required in the KATYDID core to support the rule system is an index of those units with slots of interest.

The functionality of the KATYDID rule compiler parallels that of the KEE backward chaining rule compiler — it can compile any predefined query, but is unable to handle dynamically created query forms nor deal with some of the more obscure features of KEE's rule system (e.g., alternative agenda mechanisms.) In the future we plan to extend KRICKIT to forward and mixed chaining compilers. Such compilers are no more difficult to program than the backward chainer.

4 Observations

In the development of the KATYDID system, we needed to handle several issues in the general transition from a Lisp to an ADA environment. Of primary interest among them are the topics of translating from weak to strong typing, automatic storage management, and indirectly invoked subprograms.

Issues in typing. ADA is a strongly-typed language—the data type of every object must be unambiguous at compile-time. As the experience with Pascal demonstrated, a language that is too rigid about its typing discipline is unworkable. ADA therefore includes three mechanisms for type escape: variant records, overloading, and generics. These features make the language more amenable to symbolic computing, but are inadequate for representing the complexity required by KBS. The problem with the ADA's notion of type is that it identifies type with implementation. In the real world (which is, after all, what we're trying to model) the set of "interesting" things for operations is independent their implementation. What programmers need is a facility for grouping things into classes and then checking or restricting operations on those classes, not one that starts with an overly concrete notion of class and makes it pervasive throughout the program.

Automatic storage management. Lisp programs are developed with the luxury of garbage collection. ADA places the responsibility for storage management in the hands of the programmer, who is almost certain to louse it up. We, who are trying to present a library of dynamically accessed objects to a program written with no thought of storage management have an especially hard time. We did nothing particularly clever in the

KATYDID system for storage management. However, we believe that while most applications develop considerable circular object structures, few use circular list structures. This suggest that future versions of KATYDID may be able to manage storage through reference counts.

Indirect program invocation. ADA eschews late binding of program bodies — the compiler always knows what function is called under what conditions. KEE programs, on the other hand, specifically enjoy the ability to change the active value associated with a slot, change the method in a slot, and even more grandiosely, to dynamically redefine new code through method inheritance. While the last is clearly inappropriate in the ADA environment, we have inelegantly simulated the former two by demanding the application (i.e., the application compiler) provide specific functions that map between KATYDID symbols and user code.

5 Future directions

We have developed an initial prototype of a system for translating KEE structures, rules, and programs to ADA and a prototype ADA run-time library for the use of this translated code. Our experience leads us to believe that this translation process should be possible for many applications. While the individual elements of this system have their strengths and weaknesses, considerable effort in translation, library development and the integration of the two is still needed. In Phase II of this effort, we plan to include work on the following topics.

The KATYDID core. The core is incomplete. It lacks not merely the frills like truth maintenance and graphics but important computational components like storage management and the ability to delete structures. Working with the translation mechanism, we need to resolve the appropriate data types and the occasions for their boxing and unboxing. We plan to develop a more hierarchical type structure for KATYDID components. And, of course, this critical code has not been optimized or bulletproofed.

In the long term, the ADA environment offers various opportunities for a more interesting KEE core. For example, ADA includes a multitasking mechanism, suggesting that the next version of the KATYDID core should include locking and transaction mechanisms. It may also be possible for the compiler to optimize accessing functions and data structures when it can prove suitable restrictions on the application program behavior.

The compilation systems. The compilation system can be improved in several dimensions. Within the paradigm of pure translation, we could do a much better job of type inference and type checking and employ a less ad hoc representation of ADA — discovering and representing, for example, the language's semantic constraints. As a compilation system, we also have the opportunity for optimizing the resulting system— performing functions such as data structure optimization, synthesizing variants of core library modules for particular varieties of applications and dead code elimination.

Viewed from a larger perspective, there is considerable opportunity in this work for improving the overall quality of knowledge-based applications. We expect the ultimate

KATYDID system to include utilities for interactive dialog with the developer about the hidden assumptions and behavior of the application, mechanisms to ensure that an application stays within the subset of KEE the core supports, verification and validation mechanisms to critique the structures and programs of the application, and an integrated solution of the problem of system maintenance in a translation environment.

Acknowledgments

We would like to thank the National Aeronautics and Space Administration, Marshall Space Flight Center for their support of this work under NASA Contract NAS8–38036. KEE is a registered trademark of IntelliCorp Inc. ADA is a registered trademark of the U.S. Government, AJPO. Refine is a trademark of Reasoning Systems.

References

[1] Balzer, R., "Transformational Implementation: An Example," *IEEE. Transactions on Software Engineering*, v. 7, no. 1, January 1981, pp. 3–14.

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

[3] Fikes, R., and Nado, R., "Saying more with frames: slots as classes," Working Notes of the 1989 AAAI Spring Symposium Series, Stanford, California, 1989, pp 10–14.

[4] Fikes, R., Nado, R., Filman, R., McBride, P., Morris, P., Paulson, A., Treitel, R., and Yonke, M., OPUS: A New Generation Knowledge Engineering Environment. IntelliCorp, Mountain View, California, 1987.

[5] Filman, R. E., "Reasoning with worlds and truth maintenance in a knowledge-based programming environment," *Communications of the ACM*, v. 31, no. 4, April 1988, pp. 382–401.

[6] Filman, R. E., Bock, C., and Feldman, R., "Compiling Knowledge-Based Systems Specified in KEE to ADA," Phase I Report, IntelliCorp, Mountain View, CA, 1989.

[7] Green, C., and Westfold, S., "Knowledge-Based Programming Self-Applied," *Machine Intelligence 10*, Chichester, Ellis Horwood Ltd. Halsted Press, 1982, pp. 339–359.

[8] Malin, J. T., and Lance, N., "Processes in Construction of Failure Management Expert Systems from Device Design Information," *IEEE Transactions on Systems, Man and Cybernetics*, Vol. SMC-17., No. 6, November 1987, pp. 956–967.

[9] Smith, D., Kotik, G., and Westfold, S., "Research on Knowledge-Based Software Engineering Systems at Kestrel Institute," *IEEE. Transactions on Software Engineering*, v. 11, no. 11, November 1985, pp. 1278–1295.