SKILL™: A CAD System Extension Language

Timothy J. Barnes

Cadence Design Systems, Inc.
555 River Oaks Parkway
San Jose, California 95134

Abstract

SKILL is a programming language that supports both command entry and procedural customization in Opus™ Design Framework™. After briefly considering some related work, we examine the requirements that motivate the provision of a programming language available to the user and describe some of the technical characteristics of the language design and implementation. Finally, we describe our experience with the language and outline future work. A number of programming examples are appended.

Introduction

The design of a modern system for Electronic Design Automation (EDA) is motivated by two opposing concerns. The first of these is *efficiency*. Layout, synthesis, and simulation algorithms, for example, are dominated by concerns for efficiency. On the other hand, it is critical that the system be *flexible* because changing implementation technologies, design tools, and methodologies require rapid customization.

We use the term *extension language* to refer to a programming language that explicitly supports extensibility of the CAD system.

This paper describes one such language. SKILL is designed to be invisible until needed while still providing the user with a level of tool and design control that cannot otherwise be achieved.

SKILL has been phenomenally successful both within the company and for our customers. Programs of over 30,000 lines in length are in current use, ranging in function from netlisters to module generators to a Prolog interpreter implemented in SKILL. SKILL is widely used both as a rapid prototyping tool and as the principal implementation language for a number of products.

Related Work

All CAD systems provide some facility for customization: Calma GPL [Sm75] and AutoCAD AutoLisp [SR89] are characterized below as typical examples. However, there are many other CAD system extension languages that we do not have space to consider: for example, EDA Systems Inc. E-machine [BG87] and the Berkeley CAD tools [Ha86] VemLisp.

Calma GPL

GPL is the extension language for the Calma GDSII product. Its syntax is Algol-like, and it functions both as the command language and as a programming language. GPL provides both read and write access to the database and supports an interface to the underlying programming language in which GDSII is implemented. Both functions and procedures are available, but they are limited to a maximum of two arguments.

GPL is somewhat limited in data management terms, providing no garbage collection.

AutoCAD AutoLisp

AutoCAD AutoLisp language provides both facilities for command definition and extension language programming. AutoLisp also provides special constructs for command definition, which defines syntax independent of the underlying Lisp language. This means that there is a difference between extension language programs and commands, though both may be implemented in the same language.

Requirements

We can divide the requirements for an extension language into four major areas:

- Command definition and modification
- Control structure
- Access to CAD system services
- Environmental requirements

Command Definition

CAD software is extremely complex. The command set is inevitably large. In different circumstances, one may wish either to disable some commands, to provide simplified versions of commands, or to add new commands built either from previous commands or from more primitive system functionality.

Most systems support some mechanism through which commands can be modified. In Opus, however, SKILL is not only a fully featured programming language; it is also the command language. This has the obvious advantage that the user has to learn only one language in order to execute commands, define new commands, or write a program to traverse the database, for example. This appealing simplicity, however, fails to account for the different syntactic requirements of an interactive command language and a full-fledged programming language.

In detail, the requirements for a command language are the following:

- **Very simple syntax:** While modern CAD systems do not require users to type commands in general, there are occasions where this input method is used by experienced users. A simple, clean syntax is a valuable productivity aid.
- **Support for graphical input:** It should be possible to combine textual and graphical (pointer-based) command input *in the same command.* This requires clean integration of graphical event handling with the command interpreter.
- **Accelerator support:** Facilities to bind short combinations of key-press and pointer-click events to commands offer a valuable speedup of command entry.
- **Macro generation:** The ability to group a number of consecutive commands into a single new command provides a simple kind of customization that is particularly appropriate for repetitive operations.
- **On-line help:** Ideally there should be some automatic link between command entry and on-line help. It should be possible at any time to find out how to proceed with the current command or to find out which commands are available for some purpose. Context-sensitive help facilities require that the help system be integrated with the command interpreter.

It is important to recognize, that although all command entry ultimately takes place through the command interpreter, commands originate there infrequently. The command interpreter is backed up by an extensive set of user interface facilities, managing menus, forms, dialog boxes, and a variety of editors, all of which generate commands in response to user interaction.

Control Structure

To support applications of useful complexity, the extension language must provide a full set of control structures. A minimum set includes procedural abstractions (function definition), control flow operators (**if, while**), iterative constructs such as **for** loops, the ability to perform recursive evaluation, and a mechanism for clean recovery from errors. Error recovery is a particularly difficult problem when the extension language is integrated with the underlying implementation language.

Access to CAD System Services

In order to be useful, an extension language must provide access to all the facilities of the CAD system. This includes the capabilities of both the CAD framework and those of the tools. It should be possible to write a single program in the extension language that makes use of user interface facilities, accesses the design database, and invokes primitive capabilities of one or more CAD tools.

The extension language should also provide access to operating system facilities and, through some form of Inter-Process Communication, offer an interface to CAD tools and other applications running in another address space.

Finally, the extension language should provide access to the meta-data associated with design methodology management and the version and project control aspects of data management.

A secondary, but very important requirement associated with access to CAD system services is the provision of appropriate datatypes to model the objects manipulated by the CAD tools. This includes window objects and database objects, for example. This suggests the provision of facilities to map new objects into the extension language, perhaps through an interface to the underlying implementation language.

Environmental Requirements

In addition to the functional requirements, we identify a number of requirements that have to do with the environment in which the extension language is used:

- Access to the extension language from the underlying implementation language
- Ability to register application functions with the extension language to provide new extension language functionality
- Garbage collection
- Debugging and trace facilities
- Performance analysis tools

The Structure of SKILL

SKILL is built upon a dynamically scoped Lisp interpreter fashioned after Franz Lisp [FS83]. We chose

an interpreted language because it is easier to provide a supportive development environment with an interpreter than with a compiled language. In addition, our software is ported to many different machines, and we were unwilling to undertake the development and maintenance of code generators for an unknown set of current and future architectures.

Run-time speed was not one of the requirements listed in the previous section. Obviously there comes a point where a piece of software runs so slowly that it is impossible to undertake tasks of useful complexity; however, it is our experience that most programmers are not limited by the interpretive nature of the language. The decision to use Lisp was motivated by the very simple evaluation model of that language, combined with the great flexibility and extensibility it provides. We endeavor to hide much of the syntactic flavor of Lisp behind a custom-built parser that supports an expression syntax reminiscent of C, the language with which CAD people, if not design engineers, are most familiar.

We do not believe that C programming language [KR78] is an appropriate extension language for several reasons. First, C is not an extensible language. It provides procedural abstractions and support for complex data structures, but no support for the creation of new control structures. Second, C syntax is extremely complex. This is not appropriate for either a command language or an extension language to be used by nonexperts. Finally, of course, a C compiler is available on the platforms we support, and interface libraries are available for those who wish to go to the trouble of writing C programs. Such a choice is motivated by concern for efficiency rather than flexibility.

The choice of Lisp as the basis for our extension language immediately meets a number of the requirements:

- An interactive *read/eval/print* loop that can be used as a command input mechanism
- A full set of control structures as specified above
- An intrinsically extensible language, supporting not only the addition of new functions, but also the provision of new control and data structures
- Automatic memory allocation and garbage collection

In addition to the list above, SKILL provides the following:

- A broad set of operating system functions
- An interactive debugger
- Trace and performance analysis facilities
- A function registration mechanism by which new commands can be added
- Access to SKILL functions and data structures from within C programs
- An inter-process communication protocol through which bidirectional communication is supported between SKILL programs and applications running in a different address space
- An integrated error-management strategy that works across the C - SKILL boundary
- A system for creating new datatypes

Types and Type Checking in SKILL

One way in which SKILL differs from other Lisp-like languages is in the provision of automatic type checking. This is a safety net that simplifies application code by automating a number of run-time checks that would otherwise need to be coded explicitly.

The primitive types supported by SKILL include the following:

- integers
- floating point numbers
- lists
- symbols
- strings
- ports (a stream-like abstraction for input/output)
- window objects
- database objects

Each object type has a special letter associated with it that is used to provide type information in function definitions and also as a format character for formatted input or output.

In addition to the built-in types, we provide a "user-defined" type system. This allows programmers to encapsulate new objects within the SKILL environment for specific purposes. These are implemented in an object-oriented manner. Their internal structure is hidden from SKILL, but a set of "methods" is provided to support standard operations such as field access, initialization, and comparison.

Special Syntactic Facilities

Because we use SKILL both as a command language and as a programming language, we have developed a parser that is much less restrictive than the usual Lisp parser. Much of the additional syntax is similar to operators in the C Programming Language; however, special syntax simplifies some CAD-specific operations. The reason for providing C-like syntax is simply that designers and CAD engineers tend to be more familiar with C than with Lisp.

In summary, our goals in providing a "new" syntax were as follows:

- Eliminate Lots of Irritating Silly Parentheses, thus simplifying typein

- Provide some of the special C functions to which CAD developers and integrators have become attached
- Provide simple notations for common CAD-specific operations
- Support infix representation of arithmetic and other operations
- Continue to support the full Lisp language for the experienced programmer

It is important to note that all the new syntactic forms are generated through the parser, and that the output from the parser is actually "normal" Lisp forms. We do, however, provide a pretty-printer that formats the internal representation in a manner consistent with the input.

Syntactic Examples

By providing a new syntax layered on top of Lisp, we were able to avoid designing a new language from scratch, while providing some specific benefits to the CAD user who is not familiar with Lisp.

Let us examine some of the special syntactic forms.

```
> x = 4.0 * sin( a + b * c )
```

This assignment statement is parsed into the following Lisp form, which is then executed in the normal way:

```
(setq x (times 4.0 (sin (plus a (times b c)))))
```

Either form is legal input; most users prefer the former, however.

```
> a.b
> a.b = c
```

The dot operator mimics C structure accessors by mapping to a property list function. In the first case the resulting expression would be **(getqq a b)**. In the second, the entire expression becomes a single call as follows: **(putpropqq a b c)**. These functions require that the first element (**a** in this case) be a symbol, and the property list involved is that of the symbol. We use property lists in SKILL to model C structures with named fields.

```
> a->b
> a->b = c
```

The arrow operator functions similarly to the dot operator; however, the symbol on the left is evaluated and its result is required to be a property list. We call these property lists that exist as symbol values *disembodied* property lists, or *dpls*.

```
> a~>b
> a~>b = c
```

The "squiggle-arrow" operator is a more sophisticated version of the property list functions previously described; it is CAD-specific in the sense that it applies only to database objects. The advantage of this operator is that the left-hand side can be a list of database objects, in which case the function is automatically mapped across the set. This provides a simple kind of automatic iteration that is particularly useful. An example is provided in the appendix.

```
> n:m
```

The colon operator is used as a simple way to define a coordinate pair. **n** and **m** may be either constants or expressions. The result of parsing, however, is a list of two expressions, which upon evaluation becomes a list of two numbers.

```
> n++
```

The postincrement operator provides the familiar C behavior. We implement a special function that returns the original value of its argument in addition to incrementing the variable's value.

```
> a == b
```

This operator is translated into the lisp **equal** function. This is not necessarily the most efficient function to use. If the arguments are atomic symbols, for example, the **eq** function would be better. This is an example of a minor efficiency loss due to the special syntax. The SKILL == operator, however, compares strings and other SKILL objects, so it represents a good compromise with respect to our goals of simplicity and flexibility.

Command Aliases

In addition to the above facilities, we provide an aliasing mechanism that allows a user or programmer to associate a new name with an existing command. This is a simple way of reducing typing. For example, the following expression, which may be typed in directly or included in a startup configuration file, allows a clock-watcher to type **GT** instead of **getCurrentTime.**

```
> alias GT getCurrentTime
```

The C - SKILL Interface

One very important objective in implementing SKILL was to provide the developer with a real alternative to C when creating new applications. In order to make this work, it has been necessary to support the execution of SKILL code from within C programs and the binding of C programs into the SKILL world. These interfaces are not without technical challenges. In particular, the management of errors (error recovery causes drastic changes in the execution stack) and the handling of garbage collection require special care from both the SKILL development team and the mixed-language programmer.

Registering C Functions to SKILL

The interpreter is implemented in C, so a single mechanism can be used both by the SKILL development

team and by the application developer to bind C functions to SKILL function names. The rule is that C functions that are callable from SKILL take a single argument, which is a SKILL list. They also return a result of the same type. We provide functions to disassemble these lists from within the C routine.

When registering a function, the programmer provides the following information:

- SKILL function name
- C function name
- Function type (lambda, nlambda, lexpr, macro)
- Minimum number of arguments
- Maximum number of arguments
- Argument types

Writing SKILL in C

C programmers within the company are able to access SKILL data structures and to execute SKILL functions directly from their C code. A library of functions is provided to construct and access SKILL data structures and to evaluate SKILL expressions.

There are two difficult problems to solve in the construction of this kind of interface:

- Error handling
- Garbage collection

Error handling is difficult because at any time we may have a mixture of SKILL and C calls on the execution stack, and error recovery frequently involves a long jump back to a known state of the evaluator. While unwinding the stack, it is necessary to ensure that data structures created by both the C and SKILL worlds are properly freed. SKILL handles this with a hierarchy of error handlers, which may be implemented in either SKILL or in C. When an error occurs, these handlers examine each stack frame, performing any necessary cleanup.

Garbage collection is an equally difficult problem. When the garbage collector is invoked, it marks all objects known to it. In practice this means all global symbols and their contents and variables on the SKILL execution stack. This does not include objects that may have been allocated by C programs and stored in C variables, perhaps during an incomplete calculation. We solve this problem by a variety of protection mechanisms that allow the C programmer to explicitly place protect pointers on the SKILL stack to ensure that temporary SKILL objects are properly marked before the sweep phase of the garbage collector is invoked.

The protection discipline must currently be manually maintained, and failure to follow the rules can lead to subtle bugs. A number of run-time tools help isolate these problems during software development.

Usage and Experience

SKILL is widely used within the company. In particular, *all commands in the Opus system are available in SKILL*. This means not only that the interface to all CAD system functionality is available in a single consistent way, but also that the programmer has available all the resources of our software system when constructing new applications.

Applications within Research and Development include the following:

- Command definition and interpretation
- User interface implementation and customization
- Implementation and customization of primitive editor functionality
- Parameterized cell description
- Structure compiler block and topology description
- Module generation [LW86]
- Analog system customization
- Design Flow Management
- Translator customization.

SKILL is also heavily used by our Application Engineers. Their main uses of the language have to do with extending the existing functionality by building relatively small programs to do things like repetitive database modifications, interfaces to new tools, custom netlisters, and backannotation.

That any one of these functions is supported within a CAD system is not surprising; however, we believe that the use of a single language for all of these functions is a major achievement. The obvious benefit to the end user of such a unified environment is a reduction in the cognitive load associated with learning a large number of tools.

SKILL provides a level of flexibility, integration, and consistency within the Opus design system that we believe is critical to our success in a rapidly changing marketplace.

Future Work

The success of SKILL has led to a number of requests for enhancements. Of these, the request for a compiler is by far the most persistent. We regard this as an endorsement of the language design because it implies that users would prefer to write more performance-critical code in SKILL rather than in C. Because the syntax of the language is separated from the internal representation in a clean way, it is relatively straightforward either to implement a compiler on top of the existing engine or to replace the engine with either a Scheme [SS75] or Common Lisp [St84] implementation.

We would also like to improve the C-SKILL interface, eliminating some of the tricky details of storage management, and to provide a package system similar to

that provided by Common Lisp. Name space control is an important issue in a system where all the functionality of all tools is available from a single command environment.

Conclusions

SKILL is an absolutely central component of our Design Framework product. Providing both a command language and a fully developed programming language with access to all the tools and services contained in the Opus design system, it is the primary mechanism for system customization. Perhaps the best indication of our confidence in the language comes from our customers: there are an estimated 1,000,000 lines of SKILL code in use throughout the world.

Acknowledgments

SKILL was originally developed by Larry Lai, Graham Wood, and Steve Law. Of the many people who have contributed to its success, special thanks are due to Carl Smith, Ed Petrus, and Ken Friedenbach.

Appendix: Code Samples

The following examples exhibit only a few of the capabilities of SKILL. They are intended to give the reader a flavor of the language.

Command Entry

The following expressions are all legal commands in SKILL. The parser contains some simple heuristics to distinguish functions from variable references.

```
sin 34.0
getCurrentWindow
x = getCurrentWindow()
(getCurrentWindow)
printf "Hello World\n"
setSkillPath( strcat( "/tmp"
                      getSkillPath ))
```

Observe that the functional style of SKILL allows nesting of command and procedural forms in a natural manner.

Database Access

The following expression generates a list of the names of all the instances in a given cell. It works as follows: Cell is a database identifier. It has a property called instances that returns a list of the database identifiers of its instances. Instances have a property name, and the automatic mapping capabilities of the ~> operator causes the name property to be accessed for all instances in the cell.

```
cell~>instances~>name
```

Next is an example that generates a list of all the objects with a certain property, adds them to the selected set, and returns the list. This is a typical small SKILL application that might be written by a CAD engineer or designer.

```
procedure( findProp( name )
  let( (shapes)
     shapes =
       setof( s getEditRep()~>shapes
              s~>STREAM\ PROPERTY\ 1
                                 == name )
     if( shapes then
         selectObject( shapes ))
     shapes))
```

Factorial Function

There are two definitions for the factorial function. First, the recursive definition:

```
procedure( factorial(n)
 if( (n==0)
    then 1
    else n*factorial(n-1)))
```

Now the iterative definition:

```
procedure( factorial(n)
  prog( (f)
    f = 1
    for( i 1 n (f = f * i))
    return( f ]
```

In the second example, the "super right bracket." This simple device automatically enters enough right parentheses to correctly complete the current expression (in this case the procedure definition).

References

[BG87] Brouwers, J. and Gray, M. "Integrating the Electronic Design Process," VLSI Systems Design, June 1987.

[FS83] Foderaro, J. K., Sklower, K. L. and Layer, L. "The Franz Lisp Manual," Chapter 6, Unix Programmers Manual Supplementary Documents, 1984.

[Ha86] Harrison, D., et al. "Data Management and Graphics Editing in the Berkeley Design Environment," Proceedings of the IEEE ICCAD-86, 1986, pp 20-24.

[KR78] Kernighan, B., and Ritchie, D. The C Programming Language, 2nd Edition, Prentice Hall, Englewood Cliffs, NJ, 1978.

[LW86] Lai, L., and Wood, G. "Skill - An Interactive Procedural Design Environment," Proceedings of CICC '86, pp 544-547.

[Sm75] Smith, C. "Calma's GPL™ Language- A Programming Language for Custom, Turnkey Graphic Systems," Proceedings of the Fall 1975 IEEE Compcon, 1975.

[SR89] Smith, J. and Gesner, R. Inside AutoLISP, New Riders Publishing, Thousand Oaks, CA, 1989.

[SS75] Steele, G.L. and Sussman, G.J. "Scheme: An Interpreter for the Extended Lambda Calculus," Memo 349, MIT Artificial Intelligence Laboratory, 1975.

[St84] Steele, G., Common Lisp: The Language, Digital Press, 1984.