SKILL: a Lisp Based Extension Language

Edwin S. Petrus

Cadence Design Systems, Inc.
555 River Oaks Parkway,
San Jose, CA 95134
esp@cadence.com

This paper describes an experience with Lisp as an extension language for a large electronics CAD environment and the role it plays in software design automation. This paper is not about extension languages in general, for an analysis of extension languages in CAD, see, [HNS90] and [Bar89]. Cadence is a full range supplier of software based Electronics CAD tools.

SKILL has been leveraged for various uses throughout the product range and the software development cycles. Among the important roles SKILL plays:

- An environment for software development
- A meta-language for describing domain specific languages
- Extension Language for Product customizing
- Test automation

The paper goes on to describe each of these roles, and more, after a brief introduction to the history of SKILL.

SKILL's heritage

Skill was originally conceived as a base language for developing simulation and test description languages. The acronym was adapted from the initials for Structure Compiler Interface Language (SCIL) and changed to the present and more pronounceable version.

Lisp was chosen as a base language because it lent itself well to interpreter based implementations and Lisp is a natural for code generation and as a meta-language. The first implementation was an interpreter based on the semantics of Franz Lisp [FSL83] developed at Cadence Design Systems in the mid-1980s.

The popular syntax for SKILL is a C-like alternative to Lisp. When SKILL was first introduced, the connection with Lisp was not promoted in electronic engineering circles. The SKILL parser can actually handle both Lisp and the alternative C-like syntax. With the flip of a status switch all pretty printing functions can write either in Lisp or the alternative style. For example, the two expressions below have the same internal representation and thus are interchangeable.

```
(defun myFunc (x)
    (let (y)
        (setq y (plus x 1))
        (printf " inc(%d) = %d\n" x y)
    )
)
procedure(myFunc(x)
    let((y)
        (y = x+1)
        printf("inc(%d) = %d\n" x y)
    )
)
```

Since the first implementation of SKILL, the various Lisp dialects converged on Common Lisp (CL) [Ste90], but because of the large installed SKILL base it has been difficult to move SKILL in a similar direction. Subsequent additions and enhancements to SKILL borrowed from CL and C. For instance, the I/O and string handling functions were based on C libraries; function arguments can use keywords, optional and rest designators - from CL; etc.

In general, convenience, ease of use and robustness are the main guiding forces behind the evolution of the SKILL language.

The set of basic types supported in SKILL includes: integers, floats, chars, lists, symbols, strings, function objects, defstructs, arrays, hash tables, ports, external types (C structs). SKILL also supports a subset of CLOS.

SKILL's implementation relies on a byte-code compiler/ interpreter. The memory manager deploys a conservative mark and sweep style garbage collector.

The Architecture

The SKILL interpreter is usually found at the heart of an executable. By "executable" we mean an all encompassing program containing, possibly, many applications. The interpreter is used as the basis of the command language of the applications (in most cases the command language is straight SKILL). If a graphical user interface (GUI) is present, the interpreter is used to execute the call backs from GUI events.

The GUI and inter-process communication facilities rely on SKILL's ability to carry several execution threads simultaneously. The current implementation is not fully multi-threaded. But, by using a simple mechanism for nesting threads, the basic ability to block the execution of certain threads while leaving the system active to respond to GUI or inter-process activity has been achieved.

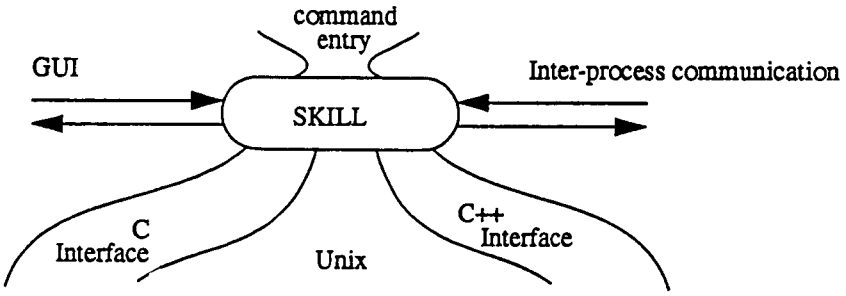
Applications export functionality from C and C++ modules to the SKILL level. Thereby allowing SKILL programs access to the underlying functionality and data of the various applications. Application functionality is exported by making C structures and procedures accessible from SKILL. To the SKILL developer, functions exported from C look and feel like regular SKILL functions.

The encapsulation of C structures allows applications to tap into SKILL's generic functionality, enabling C structures to look and feel like SKILL's internal types. For instance, SKILL's syntax supports the generic "->" operator, (e.g. y = x->fieldName) and in SKILL it has the same connotation as in C: structure access. The generic nature of "->" means that encapsulated C structs will accept the operator and get/set the value of the fields. For instance one important use of this feature is in exporting persistent database objects to be manipulated directly by SKILL programs; if the global variable x is bound to an object from a graphics database, a simple operation like,

```
x->color = 'green
```

can have the side effect of dynamically changing the color of the object (on the screen if the object is displayed). This change of property value can be reflected into the database immediately, making it available to anyone with a read only access on the same database. The "->" operator applies to other data structures such as association lists, property lists, defstructs, hash tables, etc.

It is the collective set of functionality and data exported by applications to the SKILL level that enriches the basic SKILL language and allows developers and customers alike to launch new or customized applications that would otherwise have required greater time and effort to develop.



The general architecture of embedded SKILL

SKILL as a development environment

The basic language offers a rich spectrum of data structures and library calls, augmenting those exported from the applications, offering application developers a rich set of reusable building blocks.

The interpreter coupled with an integrated editor can instantaneously reflect code changes in the runtime image of the executable. Cutting on the long edit-compile-link cycles for large, statically linked executables.

One of the earliest development tools added to SKILL was a "dbx" like debugger (dbx is a widely available debugger in UNIX environments). This offers basic debugging capabilities like setting break points, stepping through expressions, displaying stack traces, etc. Tracing function calls prints values of the arguments and the returned result from functions. In general, most of the debug features can be conditional.

SKILL developers are prolific writers. With a large volume of SKILL code developed and maintained came the need to unearth performance bottle necks. A profiler was developed for profiling time spent in procedures and memory consumed by procedures. Developers not well versed with Lisp's appetite for cpu and memory resources rely on the profiler to detect these performance bottle necks.

Given that large volumes of SKILL code is written by Cadence developers and customers, it became important to provide tools for measuring the adequacy of quality assurance tests. A tool for measuring test coverage in SKILL was developed (in the same vain as C "tcov"). This tool traps the execution of SKILL expressions during a given test run and generates statistics about the proportion of expressions in a set of modules that were exercised. This tool can reproduce detailed output of specific modules outlining the exact number of times each expression was exercised.

For the most part, SKILL developers come from a EE background and lack general familiarity with functional languages. As a result, company wide code inspections revealed the need for a high level code analysis tool that can be used by individual developers as an aid toward writing better code. This tool, called SKILL lint, was designed to take as input a set of SKILL files and to produce a set of recommendations. The rule set for SKILL lint is extendible. The rules set reflects intimate knowledge of the current SKILL implementation and aims to steer developers away from costly operations and to encourage or promote the use of the more obscure but powerful features of Lisp.

A code browser was added to the development suite to aid developers in tracking and understanding control flow of SKILL programs. This information is presented as a DAG. The browser is also used to display profiling information.

One of the chief strengths of SKILL as a development environment is its robustness. Most of that can be attributed to the abstraction over the address space afforded by an automatic memory management system. It is interesting that we should mention garbage collection in the section on software development, but that is one of the most cost saving and quality enhancing aspects of SKILL. It has been our experience that in large software systems, tracking memory related bugs (core leaks, memory over-writes, etc.) is one of the most costly activities for a software organization. Automatic memory management eliminates all that and frees software engineers to concentrate on the algorithms and logic - the meat of the problems. Because of this, the general theme at Cadence is that code written in SKILL can not cause crashes.

During SKILL's brief history, special attention was paid to aspects that help developers avoid making mistakes. For instance, we find the generic nature of certain operations to be useful. Consider, "foreach", SKILL allows this operator to accept as argument lists, hash tables, C structs, etc. The syntax for accessing arrays, e.g. arr[index] = value, is allowed to accept other structures such as hash tables and lists. In the case of hash tables, say, the variable spouse was bound to a hash table, then,

```
spouse['John] = 'Jane
```

creates an entry in the table with John as the key and Jane as the value.

As insignificant these extensions may seem, they help create familiarity and steer developers away from making inadvertent programming errors. This simplicity and generic nature of operations makes it easy for developers to experiment during an implementation with different forms of data representation. It also promotes frequent re-writing of existing code to improve maintenance and performance. A practical way of dealing with the syndrome of productizing prototypes.

SKILL as a meta-language

SKILL's original purpose was as a meta-language for defining and implementing domain specific languages. For example, there are simulation, test generation, procedural layout languages available, all written on top of SKILL. For the most part these domain specific languages look and feel like regular SKILL with added macros and syntactic extensions (meta characters) to serve their specific purposes. Lisp as a meta-language is old news to experienced Lisp users. But in electronics engineering circles it is, to say the least, a revelation.

Another use of SKILL, that may not necessarily qualify as meta-language but nevertheless is worth mentioning, is for embedding procedural "knowledge" in design objects. For instance, certain attributes of an object need to be computed dynamically when accessed. Database managers call on SKILL to evaluate these expressions at run time. This service, standardizes and simplifies the implementation of procedural knowledge/information across the product line.

Product Delivery and Customizing

SKILL offers a unique vehicle for packaging and delivering products written in SKILL. The basic idea revolves around the simple notion of dynamic loading and linking of predigested SKILL code. Products are divided into capabilities and for each capability that has corresponding SKILL code there is a *context* file. A context file is a binary representation of the runtime image of that particular capability - containing code and data. For instance, the capability to view schematic designs can be separated from the capability for editing schematic designs, therefore, users intending to read schematics without altering their contents load only the code relevant for viewing. Code enabling editing is left out.

The context file is generated during an integration phase. Basically, a capability is allowed to load all of its code and generate all the setup data it needs. This the point at which the capability is ready

to be used. This is also the point at which the context file is generated. At runtime, when the capability is called upon by a user to perform, the context file is absorbed and the capability is reinstated ready to be used. Essentially regenerating the state saved in the context file during integration.

By incrementally loading context files, capabilities are loaded only when needed. Loading context files is quite fast since all that is being loaded is a snapshot of a portion of the heap. The disruption to the user during a context load is minimized. With the right break up of a product into capabilities, only the needed subset of the entire code for a product is likely to be loaded in a session.

Customizing can be achieve by allowing users to register call-backs for any capability they wish to customize. This call-back is in the form of a zero argument procedure that is called immediately after a context file is loaded. During the call-back, user code can add to the functionality of the capability by loading and executing custom code. For instance, new menus or fields on existing menus can be added, forms can altered, databases can be modified, etc.

SKILL in Test Automation

Because most applications export a good deal of their functionality through procedural interfaces at the SKILL level, it is possible to write explicit tests in SKILL to exercise the underlying functionality. These tests include unit tests, functionality coverage tests and regression tests. Using the SKILL test coverage measuring tool mentioned earlier developers and managers can measure the adequacy of the tests.

In the spirit of the theme that code written in SKILL can not crash, testing the functionality exported from C into SKILL to ensure integrity is critical. SKILL can't crash but code written in C or C++ can.

It was decided that we should automate the generation of tests as much as possible. The first target was the testing of boundary conditions. This attempt was mitigated by the fact that all C wrapped SKILL procedures declare the types of their arguments. Given that, we wrote tests that generate "randomized" calls to the C wrapped procedures to test their boundary conditions. For instance, if a C wrapped procedure expects an integer as an argument then calls to the procedure are generated that use, say, -1, 0, 0x7fffffff, etc. This can get fairly complex and elaborate given the nature of the types, but it does potentially eliminate a source of a large number of easy to fix crashes. Needless to say this is a popular test in quality assurance groups.

SKILL for novice users

In may instances, those who learn SKILL have had little or no programming experience in their past. Their interest is to just learn enough to get by. Usually their first exposure to SKILL will be in an introductory level class given by Cadence on the subject. The core language is the same in all products but the application PIs can vary. So, one of the first tasks is to learn enough about the basics to start working on a particular product.

Most Cadence products offer access to SKILL at the command line of the application. For instance, a user can type at the command line of a product:

```
x = getCurrentTime(); my first line of SKILL
> "Jun 3 16:24:15 1993"
Printf("Captain's log, star date %s\n" x)
> Captain's log, star date Jun 3 16:42:15 1993
```

It is actually encouraging for a novice (and a non-programmer at that) to see that a simple start like this can be made. The dynamic nature of an interpreted environment is forgiving, hence users can go on to other operations all the while recovery from mistakes has virtually no cost (as opposed edit-compile-link-execute). Other and more effective operations afforded by the abstractions over the various subsystems are just as simple. For instance, an abstraction over UNIX inter-process communication mechanisms lets users customize the application to send email from the application:

```
ID = beginProcess("mail user")
writeChild(ID "Beam me up Scotty .. \n")
closeChild(ID)
```

After learning the basics, users move on to learning specific applications and their SKILL procedural interfaces.

Future Evolution

A few things are certain about SKILL's future. In the realm of compliance with standards, SKILL will migrate towards Scheme [SSu83]; although it is more likely that SKILL's semantics will adopt those of Scheme and a few features from the Scheme standard will not be supported (we know that call/cc will not be supported as it does not play well in an embedded environment). Scheme was selected as the standard extension language for CAD Frameworks by the CAD Framework Initiative (CFI - a standards body in the area of electronics CAD).

Increasing the performance of SKILL by compiling to a representation closer to native code is highly likely. Adding a good module system is highly desirable. Promoting a subset of CLOS with good coupling to object oriented databases for persistence is being driven by demands from applications.

There is a lot of room for improving the development environment. For instance, since SKILL integrates closely with the C language, it is highly desirable to have an integrated development environment for SKILL, C and possibly C++. Currently, there are options in the SKILL debugger to fall into dbx if debugging C code is necessary and macro support in dbx is available for accessing SKILL's runtime structures. But, because of the clear separation of the SKILL and C environments, there are always features in one system that are not available in the other.

Another area in need of a good solution is in the realm of distributed Lisp environments. There is an emerging scenario in the architecture of CAD products where multiple executables with embedded Lisp are likely to co-exist and depend on each other's services. Abstracting this multi-process environment and laying the foundations for a robust implementation is essential for the success of the architecture. The multi-process architecture for delivering products (as opposed to monolithic executables) is essential from a software engineering perspective for several reasons:

- ECAD software is too large and diverse to be contained in a single executable.

- Need "fire-walls" between products. Separating major products into separate executables reduces the chance of one set of code from corrupting data in another.
- Change control is easier when the software is built separately. That is, a change in one product does not necessitate testing all products residing in the same executable to ensure integrity.
- Independent release of products. Products linked together in the same executable must be shipped together. In separate executables products can be shipped whenever necessary (provided interface compatibility is maintained).

It has been the experience at Cadence that Lisp adds a great deal of value to the process of developing complex software. All the uses of SKILL described in this document make one point clear: Lisp the language and Lisp environments are valuable for engineering software. It is a pity that Lisp is so synonymous with artificial intelligence (AI) and that brings with it a perception of failure (at least this is the case in disciplines far removed from AI).

Software engineering is certainly one area where this paper claims Lisp can add value. But it is an important area. If Lisp is to continue to be of value in software engineering it must continue to offer more than conventional software engineering tools. It is also importance for Lisp to do away with its weak points if it is to be widely accepted.

For instance, the size of CL was the main reason that CFI dismissed it as a standard for CAD extension languages (even though it was acknowledged that CL is the only viable professional Lisp in the market). Scheme was the only other alternative with a standards body. In ECAD, it is expected that an embedded Lisp library should not weigh more than 500Kb. Most will say less is desirable and some will tolerate more. If we consider that a professional schematic editor can be written to weigh less than 2Mb (including graphics, operating system libraries, etc.) - one can begin to see a formula. Other than size, CL is a very powerful language with more features than an average developer is ever likely to need. The current professional implementations of CL are certainly state of the art and are very competitive in performance (w.r.t. environments for other languages).

Software engineering needs the kind of automation that ECAD tools brought to bear on integrated circuit and systems design and launched them into levels of complexity that can no longer be managed without those very tools. Lisp and Lisp environments are ideally suited to play a leading role in such an effort. Experience with AI showed us that much can be accomplished if there is focus. The most vital asset in the Lisp community today is the available skill (no pun intended) set and talent to focus on making a positive impact on software engineering.

References

[Bar89] Barnes, T., *SKILL: A CAD System Extension Language*. Design Automation Conference Proceedings 1989.

[FSL83] Foderaro, J.,K. Sklower, K.L. and Layer, L. *The Franz Lisp Manual*. Chapter 6 in "Unix Programmers Manual Supplementary Documents", 1984.

[HNS90] Harrison, D, Newton A.R., Spickelmier R.L., Barnes, T. *Electronic Cad Frameworks*. Proceedings of the IEEE, Vol 78, No 2, Feb 1990.

[SSu83] Steele, G.L., Sussman, G.J. Scheme: an interpreter for the extended lambda calculus. Memo 349, MIT Artificial Intelligence Laboratory.

[Ste90] Steele, G.L. *Common Lisp - The Language*. Second Edition, Digital Press 1990.