Experiences with the Framework Environment Cadence Skill/IL

Andreas Koch, Ulrich Golze, Michael Schäfers

Technische Universität Braunschweig
Abteilung Entwurf integrierter Schaltungen (E.I.S.)
W-3300 Braunschweig, Germany

Abstract

This paper describes the experiences made employing Cadence's Skill/IL Framework extension language for both educational and research purposes. While the language and its facilities were examined in detail in order to evaluate it and to create a comprehensive documentation, it was also used to develop tools simplifying the use of the Design Framework for beginning students. An introductory course paper for programmers learning Skill/IL has been prepared [1].

1 Introduction

The IL language and Skill library are used to tie the diverse components of the Cadence Design Framework for VLSI design together, to automate recurring design tasks and to customize the work environment for the individual designer. IL is an interpretive language based upon C and Lisp. Many of its control constructs are borrowed from C, but the primary data structures and the dynamic concept of the language have their roots in Lisp. We report on a comparative analysis of the relative strengths and weaknesses of the language as well as the creation of comprehensive and easily accessible documentation for the prospective programmer [1].

This documentation is not designed as a reference manual, but as an introduction to the language and its facilities for the intermediate to advanced programmer. It has been written putting special emphasis on concepts such as dynamic typing, dynamic scoping, run-time creation of executable code and other aspects which are not well known to a developer versed only in traditional procedural programming languages.

2 Practical Applications of Skill/IL

In the course of evaluating Skill/IL and applying it to the VLSI semi-custom design labs offered by the department E.I.S., numerous programs have been written, ranging in complexity from a few to a few hundred lines of code. The programs described in the following address practical problems discovered while observing how students work with the Design Framework in order to solve the exercises required by the course.

2.1 Verification of File Paths - checkPath

One of the most common problems encountered when preparing a completed design for fabrication is the use of absolute path names to specify master cells. Since the directory layout and disk organization at the foundry are almost certainly different from the one used while entering the design, absolute path names cannot be used to identify component cells in a hierarchical design.

checkPath recursively checks all levels of a design for the occurence of absolute path names. Any instances on the current design level containing such names are highlighted in the GraphicsEditor, instances on deeper levels in the hierarchy are identified by listing a logical path through the hierarchy to the cell in question. In order to eliminate repeated occurences of the same error message, the tool keeps track of the cells already checked. After each custom (non-library) instance in the current hierarchy level is checked, it is queried, if a schematic master representation is available for it. If one exists, then this representation is added to the list of cells to check.

2.2 Simple Hierarchical Netlister-listNets

Another tool used successfully is a very simple hierarchical netlister. This program was originally developed only to become more familiar with the structures of the design database. However, it proved its usefulness almost immediately by discovering incorrectly routed ground nets in a design that was scheduled to be fabricated. The program has been used since to examine wiring specifics in student designs which were difficult to trace in the schematic representation. Its output consists of the instance whose nets are currently listed, the net name, its type, its signals and the actual and formal terminals connected to it together with their specifics such as name, I/O direction etc. This tool also tries to avoid unnecessary repetitions in its output by marking cells already listed.

2.3 Automating Place and Route - autoPR

The major and most useful program developed while evaluating Skill/IL is concerned with the automation of the place and route process.

This multi-step sequence of operations transforms a schematic into a layout representation. Apart from being tedious, the manual place and route process is also highly error prone, especially in a training environment. Our experiences range from incorrectly routed designs caused by forgotten global net routings to corrupted data due to incorrectly specified representation types. In order to lighten the burden for students as well as for teaching assistants, a tool was developed which simplifies the process considerably.

Only a single command has to be typed in order to initiate the semi-automatic sequencing of place and route operations. All required phases are executed in the correct order using appropriate parameters. The user only has to indicate his intent to continue execution by dismissing numerous status reports and confirmers presented during the tool run. Unfortunately, the appearance of these interactive elements prevents a fully automated run. Tool execution is suspended until the user responds to the requests. This tool highly reduces the possibility of human error during the place and route process. Since autoPR uses the openSDA interface to access non-Skill functionality provided by the Frame-

work, it was very useful in evaluating this facility (see next section).

A companion utility to `autoPR` calculates the estimated chip area of the design after placement and routing by retrieving boundary information from the design database and presenting it in an easily readable manner.

3 Evaluation Results

It is in the nature of things that many powerful features offered by the system are taken for granted and are not consciously appreciated. Errors and misfeatures, however, are immediately noted. This should be remembered if the following criticism seems overly harsh.

The creation of the introductory documentation brought disappointments as well as pleasant surprises. While desirable features seem to be missing, other language capabilities are unfortunately underrated and hidden in the standard documentation. This includes run-time creation of executable code and debugging facilities. Without prior knowledge of symbolic programming languages such as Lisp, the programmer is sure to miss many of its finer points.

3.1 Interpretive Nature

The interpretive nature of Skill/IL facilitates an incremental style of development, since the effects of program changes can be immediately observed without going through a lengthy compile-link cycle. This also enables a designer to interactively create short, specialized "one-shot programs" to simplify the current design task.

3.2 Syntax

Skill/IL syntax is easy to learn and to use. The language is easily accessible to CAD programmers. This is achieved by hiding the underlying Lisp interpreter by a parser able to understand a dialect based on C, a language most prospective users are familiar with. The application of Skill/IL as the interactive command language of the Cadence Design Framework profits from the short, concise commands unburdened by "syntactic sugar".

3.3 Dynamic Typing and Scoping

Dynamic typing and scoping combined with automatic garbage collection, also provided by the Skill/IL Lisp core, eliminate the need for many "administrative" operations such as the explicit declaration of variable types, scopes and memory organization. Since unconstrained use of these features leads to a higher risk of run-time errors, Skill/IL offers type checking primitives that can be used to verify type integrity on critical points during program execution. For example, Skill/IL functions can check their actual parameters against an argument template every time they are called. The availability of type information at run-time also makes the design of overloaded functions possible.

3.4 Interprocess Communication

The Skill/IL model for interprocess communication simplifies the integration of existing tools. While not using the more powerful OS-native IPC facilities (such as Unix shared memory and message queues), the method used by Skill/IL has the advantage that almost any existing tool that accepts or produces information using character streams (either from files or interactively) can be embedded in the Framework without necessitating changes in the tool itself.

3.5 User Interface

The Framework user interface is accessible to custom programs using Skill functions. While this user interface is not state of the art (neither in handling nor in appearance), it satisfies the requirements of an interactive session. The Skill programmer can choose from a variety of user interface components in order to present and obtain information in the most appropriate and user-friendly manner.

3.6 Design Database Access

Skill/IL provides transparent access to information stored in the system-wide design database. There is no need for specialized data definition/manipulation or query languages, all operations take place within Skill/IL by using functions and operators.

3.7 Documentation

Skill/IL suffers from an incomplete and faulty documentation which severely complicates working with the system [2]. Many of the language capabilities are not covered. For example, the creation of relations between objects in the design database using Groups as well as the Skill/Framework interface using `openSDA` remain undocumented. Even if documentation is provided, it often contains errors of a severity exceeding simple typos. Function descriptions are simply incorrect or contain an insufficient amount of information. In many cases, function effects can only be determined by trial and error.

3.8 Combination of C and Lisp

The combination of C and Lisp can sometimes lead to undesired results. The IL function `getchar`, for example, has a totally different effect than the C function of the same name.

3.9 Run-Time Modification of Code

While IL has the capability to retrieve and modify the definition of existing functions at run-time, its practical use is hampered by the fact that the majority of Framework-defined functions is write protected. This makes the customization of existing functions using "head" and "tail patching" impossible.

This restriction gains an even greater importance when considering that much of the Framework functionality exists only in the form of `openSDA` services. The granularity of these services is very coarse, they are often not optimally suited to solve a given problem.

An example is the program for semi-automatically performing the place and route process. This program controls the process on its own. However, a human user must still intervene from time to time to close status report windows whose display brings the automatic execution to a grinding halt. The information presented in these windows is often of no interest to the average user. If the single steps of the place and route process were available as "pure" finely structured IL routines, these unnecessary interruptions could be easily avoided. By calling the sufficiently specialized functions, the process could run uninterrupted. Only when an error occured, the user would have to intervene.

3.10 Error Checking

Error checking is also an aspect that suffers from the use of `openSDA` services as Framework interface. Since this function does not provide meaningful status values, it is impossible to verify the successful completion of

an intended operation. Thus, a Skill/IL program cannot recognize and act upon an error status occuring during the execution of an `openSDA` service. Inconsistently, not all of the operations necessary for the place and route process are provided as `openSDA` services, some exist as "pure" IL functions. This combination of two different interfaces can increase program complexity and decrease program legibility.

3.11 Debugging

Debugging capabilities for IL programs are adequate but not up to date. After considering that especially Lisp based environments have a long history of advanced graphic interactive debugging tools, an improvement of the IL facilities seems in order.

3.12 Run-Time Efficiency

In view of the increasing use of Skill/IL for the implementation of larger programs, the availability of a compiler would be desirable. After interactively developing and testing a program, the final version could be compiled for run-time efficiency.

3.13 String Handling

Skill/IL internal string handling is somewhat disappointing. While almost all C functions are available, Cadence discourages the use of strings due to memory management difficulties. This is no problem for programmers familiar with IL or Lisp, since equivalent functionality is provided by symbol operations. However, to better accomodate IL novices familiar with C in general and the extensive use of strings in particular, less restrictive string handling in IL would be useful.

3.14 Base Language

Also, the IL base language itself has room for improvement. Examples include a more flexible `for`-statement (able to decrement and use steps different from one) as well as extending the operations for function definitions. Specifically, the extension of argument templates to cover optional and keyword parameters could improve program robustness.

3.15 Design Database Capabilities

Apart from reliability problems, the single greatest weakness of the system are the manipulation capabilities for the design database. While dbAccess internally does have a rich repertoire of access and modification functions, the majority of these is not available to the programmer. When one of these functions is called, the system responds with an "undefined function" error message. An example for some restrictions caused by this policy is that dbAccess is able to query a signal for global scope, but unable to create a global signal using public functions, since the appropriate routines are not callable by the programmer. The implementation of the class hierarchy of the design database schema is inconsistent. For example, rectangles have attributes that are meaningful only for arcs. A consistently object-oriented approach to inheritance would have avoided these discrepancies.

3.16 Further Remarks

The whole system is not particularly robust. Many functions do not perform sufficient checks on their actual parameters. When confronted with inappropriate parameters, the range of possible system responses is wide. In the best case, functions produce unusable

results, such as invisible arcs or unreadable labels. In the worst case, the whole Design Framework terminates abnormally. Even if a function checks its parameters and discovers an invalid value, the program reacts unpredictably. While almost all Skill/IL functions are documented to return an error status as function result, many functions abort program execution immediately upon encountering an invalid parameter. This can lead to inconsistencies.

An example for this behaviour are the functions dealing with the movement, deletion and copying of objects. These functions accept a list of objects to be processed as a parameter. They handle these lists sequentially, one object at a time. If an invalid object is contained in the middle of such a list, the functions terminate immediately, leaving the design in an undefined state. Those objects preceding the invalid object in the list were processed, those following were not. Even if a function does not terminate the program on error, the return status can be unreliable, too. The `property` function, for example, returns `t` (no error), even if the operation failed.

Aside from these problems caused by inappropriate parameters, some functions cause unexpected effects even when operating with valid arguments. An example for this behavior is the function `doughnutSlice`, which, when passed a small (but legal) radius as parameter, forces the unsuspecting user at run-time to manually increase the global `#SIDES` property.

These criticisms are rather broad in nature and concern wide areas of the Design Framework. Apart from these general comments, numerous individual problems exist concerning single functions. An example for such a problem are the errors in the distance calculations of `getNeighbor`. Considering todays state of software technology, however, insufficiencies of this type seem to be unavoidable in a program as complex as the Design Framework.

4 Conclusion

Skill/IL conceptually provides powerful and flexible tools for extending and automating the framework. Many of its weaknesses can be traced to programming errors during its implementation. A careful examination of the Design Framework's Skill/IL component could be used to eliminate most of these errors without making fundamental changes in the basic system design. The language capabilities could easily be enhanced by making more of the internal functions available to the programmer and documenting them appropriately.

Bibliography

[1] Koch, A., *Integrationssprachen in VLSI-Design-Frameworks am Beispiel von Cadence Skill/IL*, Diploma Thesis, Abteilung E.I.S., Technische Universität Braunschweig, 1992.

[2] Cadence Design Systems, *Skill Language Reference Manual*, Version 2.1, 1989.