SystemVerilog, Batteries Included:

A Programmer's Utility Library for SystemVerilog

Jonathan Bromley
jonathan.bromley@verilab.com

André Winkelmann
andre.winkelmann@verilab.com

Verilab Inc., Austin, Texas

***Abstract*—As the language of choice for many verification engineers, SystemVerilog is expected to act not only as a specialist verification language, but also as a hardware description language and a general purpose programming language. Although SystemVerilog's object oriented programming features and rich set of native data types provide excellent support for general purpose programming, many users including the authors have been frustrated by its lack of utility features that would be taken for granted in other languages. In this paper we present the results of our efforts to develop a comprehensive, consistent, easy-to-use utility library for SystemVerilog. We believe this to be the first announcement of such a library that is vendor-independent and carefully tailored to the needs of SystemVerilog users.**

Keywords—SystemVerilog, DPI, svlib, utility library, document object model

I. INTRODUCTION

The project described in this paper, provisionally named *svlib*, is a collection of SystemVerilog classes and utility functions, presented as a package. It is freely available with permissive open-source licensing (see section IV.G), along with source code, documentation, simple build scripts for popular SystemVerilog simulators, and examples.

svlib complements SystemVerilog's existing capabilities by providing a variety of utility functions that the authors have found lacking in the base language. It aims to be applicable to a wide variety of programming problems, including – but not limited to – SystemVerilog's traditional "home territory" of digital design verification. Nothing in *svlib* is verification specific; those specialized features are already well supported by methodology toolkits such as the UVM[1].

As an example taken from the authors' experience, consider a DUT whose configuration is described by a file in JSON[2] or YAML[3] format[1]. The location of this file is specified by an operating system environment variable. The testbench should check that this file is no more than one day old, and then read configuration information from the file. Locating the file requires that the testbench be able to read an environment variable. Checking that the file is up-to-date requires knowledge of the current date and time, and the ability to inspect directory contents and determine a file's modification time. Finally, to read the configuration a testbench must extract information from the chosen file format, probably with further string processing to get the required information from values so obtained. The library features of SystemVerilog do not readily support such tasks, comparing poorly with general purpose languages such as C and Java, and even less favorably with scripting languages such as Python, Ruby, Tcl and Perl. Users of SystemVerilog are forced into creating their own implementations, perhaps with the help of external software via the DPI, because SystemVerilog lacks a sufficiently comprehensive library. Some facilities of this kind can be found bundled with major vendors' simulators, but these vendor-specific libraries are inherently non-portable and are not necessarily as complete as one might wish. There are also a few useful functions (notably regular-expression string matching) provided with the UVM codebase, but once again the available offering lacks the completeness that users might reasonably expect.

The paper will describe the motivations that led to the creation of this library, and our experiences in designing, implementing and testing it. Particular attention will be given to our aim of providing a SystemVerilog API that is as natural and convenient as possible. Code examples and performance measurements will be presented, allowing potential users to evaluate the suitability of the library for their own applications. The source code and build scripts will be made freely available, at the time of the conference, under a permissive open-source license. The authors hope that the user community will provide further input and refinement, making this a truly standard library.

II. OVERVIEW OF SVLIB

svlib features fall into a few broad categories, with some inevitable interaction between them. For example, functions that manipulate file pathnames have been grouped together with other file system functions, although they are in truth nothing more than specialized string operations. The available features, available to the user by a simple import of package `svlib_pkg`, can be broken down as follows:

- general-purpose string manipulations, including regular expressions
- functions for manipulating file and directory names; directory lookup, directory listing, and file property inquiries
- functions for interacting with the operating system including environment variables, command-line arguments, and wall-clock time and date
- miscellaneous utility features including convenience functions for manipulating enumerated types
- toolkit for reading and writing configuration data stored in files using `.ini` or YAML representation

[1] JSON and YAML are both file formats that provide textual representation of native data structures in a form that is both human- and machine-readable. Many other such file formats exist, for example XML.

Finally, a header file `svlib_macros.svh` defines some SystemVerilog macros providing automation of object serialization, and some miscellaneous operations that could not otherwise be offered in a convenient way.

Full details of the complete contents of the package can be found in the distribution, but the following brief descriptions illustrate a few highlights and provide some simple examples of library usage.

A. *String manipulations and regular expressions*

svlib supports full-featured regular expression matching. It offers capture of matches and sub-matches, automated search-and-replace including sub-match placeholders, and global matching and replacement (find all possible matches in a source string). A regular expression can be saved as an object, making it faster and more convenient to use for subsequent match attempts. These features of *svlib* are underpinned by the POSIX-compliant regular expression processor of the standard C library, providing a robust and well-documented implementation as described by the Linux/Unix document `man 7 regex`. We are aware that more modern and advanced regular expression processors exist, notably the Perl-compatible library `pcre`[4], and we plan to add support for this regex dialect in a future version of *svlib*.

The package also provides a range of string manipulation functions including finding a substring within a string, insertion of one string into another, splitting a string into a queue of strings on every occurrence of a separator character, and the string join function familiar from many scripting languages. Right, left and center justify functions simplify the preparation of data for display in tabular format.

Finally, a function is provided that can read numeric values in the full Verilog numeric literal syntax such as `16'hFF_6xzE`. This feature plugs a minor gap in the existing formatted I/O support.

B. *Interacting with the file system*

Traditionally it has been impractical for SystemVerilog to make file system inquiries such as determining the last modification time of a file, or obtaining a list of all files in a directory. The package provides easy access to these operations directly from SystemVerilog. For example, to get a queue of strings containing the names of all files in the current directory whose names end in `sv`, the following code fragment could be used:

```
string dirList [$];
dirList = sys_fileGlob("*sv");
```

Given this list of files, we can then easily find the most recent, with the added condition that it should have been written no more than one day ago. This example uses the `sys_dayTime` function from *svlib*, yielding the current time in the usual Unix seconds-since-1970 form, and `file_mTime`, which provides the last-modified date of any file in the same format. Finally the *svlib* function `sys_formatTime` is used to create a human-readable representation of a Unix timestamp. All these functions are implemented in *svlib* by appealing to the corresponding functions in the standard C library, with *svlib* merely providing a SystemVerilog-friendly wrapper.

```
longint mostRecentTime = sys_dayTime() - 24*60*60;
string mostRecentFile = "";
foreach (dirlist[i]) begin
  longint t = file_mTime(dirlist[i]);
  if (t > mostRecentTime) begin
    mostRecentTime = t;
    mostRecentFile = dirlist[i];
  end
end
if (mostRecentFile != "") begin
  $display("The most recent file is \"%s\"");
  $display("It was modified at %s",
            sys_formatTime(mostRecentTime, "%c");
end
```

`svlib_pkg` also includes functions to manipulate file path names, making it easy (for example) to extract the filename alone from a full pathname. We did not provide functions allowing files to be renamed, moved, and deleted directly by SystemVerilog code, as this functionality is already easily available through the `$system()` call.

C. *Interacting with the operating system*

Functions are available to find the current working directory, work with environment variables, and find the complete command line used to launch the simulator (which provides a solution to the long-standing problem of how to handle multiple plusargs with the same name).

As already noted, *svlib* also contains functions to read wall-clock time and date, and convert it to human-readable text.

D. *Configuration data files*

The package's configuration features provide the ability to read a configuration file, in `ini`[5] or YAML format, making the file's contents available so that your SystemVerilog code can read any named item from it. This will typically be used to populate configuration objects at the start of a simulation run. Likewise, arbitrary user data can be written out to a file in the same formats. Support for Comma Separated Value (CSV) and other file formats is under consideration but has not yet been implemented. The configuration file features are described in more detail in section VIII.

III. Dividing Responsibility Between Language and Libraries

A. *There's Something Missing*

Verification engineers face difficult problems daily, and it is not surprising that sometimes they express frustration that SystemVerilog does not provide all the support they need. The evident popularity of the UVM and other verification methodology base class libraries (BCLs) clearly indicates strong user appetite for ready-to-use packages of domain-specific functionality. Consequently it was surprising, to the authors at least, that there does not appear to be any comprehensive and readily available library of utility functions

for SystemVerilog. Users working in a general-purpose programming language such as C or Java, or a scripting language such as Python or Perl, would expect as a matter of course to have access to a huge and well-established collection of functions for string manipulation, file access and file system exploration, operating system interface, data structure manipulation, and similar utilities. SystemVerilog's provision of such functions is patchy: for example, text file I/O is quite well supported, but string manipulation is provided through a limited and somewhat idiosyncratic repertoire of operations, and operating system interface support is vestigial.

One possible reason for SystemVerilog's weakness in this area is that the core language, as specified by its LRM [6], already provides a useful selection of operations that in a more general-purpose language might be provided as library functions. For example, extensive formatting and I/O of textual information is available through the `$display`, `$scan` and `$sformat` families of system functions. However, having such functionality built into the core language makes it difficult to add new features, as they must be added through the protracted process of language standardization. The authors' perception is that the SystemVerilog user base has become accustomed to the idea that utility operations should be built-in, and new operations can be added only by extending the language.

This situation is at least in part to blame for the often-heard criticism that SystemVerilog is bloated and contains too many special features. One of the authors has previously argued [7] that this criticism is largely unjust, and that it is entirely appropriate for SystemVerilog to have an unusually rich set of specialized features. Statically elaborated instance hierarchy, constrained randomization, temporal assertions, and functional coverage are good examples of EDA domain-specific requirements that are better provided as core language constructs with their own dedicated syntax, rather than as library functions presented to the user through the somewhat restricted syntax of calling a task or function. On the other hand, the huge selection of library functions conveniently available to any C programmer has no counterpart in SystemVerilog, and many of those functions would be very welcome additions to a typical verification engineer's toolkit.

Each of the major simulators ships with some kind of utility library offering some of these functions. However, in every case the feature set is patchy and incomplete, and – more important – many users prefer to avoid vendor-specific offerings because commercial or technical imperatives require them to be able to use more than one tool while maintaining a common code base.

B. *Does the DPI make a library superfluous?*

SystemVerilog has comprehensive and well-standardized C-language interfaces, the VPI and DPI, described in clauses 35 and 36 of [6]. A well-known consequence of the DPI is that users can now directly call any function in the usual C library if its argument and return data types are among the set of types that are natively supported by the DPI interface mechanism. The following code example[2] shows how a standard C math library function can be made directly available in SystemVerilog using the DPI:

```
module test_dpi_sqrt;
  import "DPI-C" function real sqrt(real x);
  initial
    $display("sqrt(9.0)=%f", sqrt(9.0));
endmodule
```

No further coding effort is required. The DPI argument-passing mechanism automatically ensures that the `real` argument and return type appear as `double` values in C, and it is merely necessary for the user to ensure that the SystemVerilog simulator has access to the necessary C library object code using appropriate command-line options.

Given this very convenient access to the existing wide repertoire of C functions, it might seem unnecessary to provide any kind of utility library in SystemVerilog. However, further consideration soon shows that this is not the case. Many of the C library functions of interest depend heavily on data structures that cannot reliably be mapped directly into SystemVerilog, so some indirection is required. Furthermore, as *svlib* is ported to other platforms, some underlying C functions may change but we must preserve an unchanged SystemVerilog API.

For example, the query function `file_mTime` returns the last-modified timestamp of a file. In *svlib*, wall-clock time/dates are consistently represented as seconds since the beginning of 1970, using a `longint`. In the C library, a file's timestamp information (along with other status) is returned in a `struct stat` that must be allocated by user code. We do not wish to trouble the SystemVerilog programmer with these details, which are specific to the C library, so it is preferable to provide wrapper functions. Some of this wrapper functionality is implemented in C and some in private SystemVerilog code; the precise split is opaque to users, and may change in future revisions of *svlib*, but the user-facing `file_mTime` function can remain unchanged.

It is also important that the library as a whole should not exhibit memory leaks, *i.e.* situations in which memory is allocated but never released. This can be challenging to achieve in *svlib* because some C library functions must allocate data structures in C memory before returning that information back to SystemVerilog. After returning control to SystemVerilog, the C code is no longer running and has no way to deallocate the memory that it needed. In *svlib* we handle this problem by careful management of the shared responsibility between SystemVerilog and C, so that the C code is always given a chance to deallocate any memory it has used. This can be done only if a strict protocol is established between the C and SystemVerilog parts of the library, and this in its turn implies that there must be some parts of the library infrastructure that remain hidden from users so that their integrity is assured. The overall library architecture then appears as in Figure 1.

[2] Since the 2009 revision of its language standard SystemVerilog has supported a wide selection of math routines, including `$sqrt`, as predefined system functions. Consequently this example is no longer useful, but it serves to illustrate the principles involved.

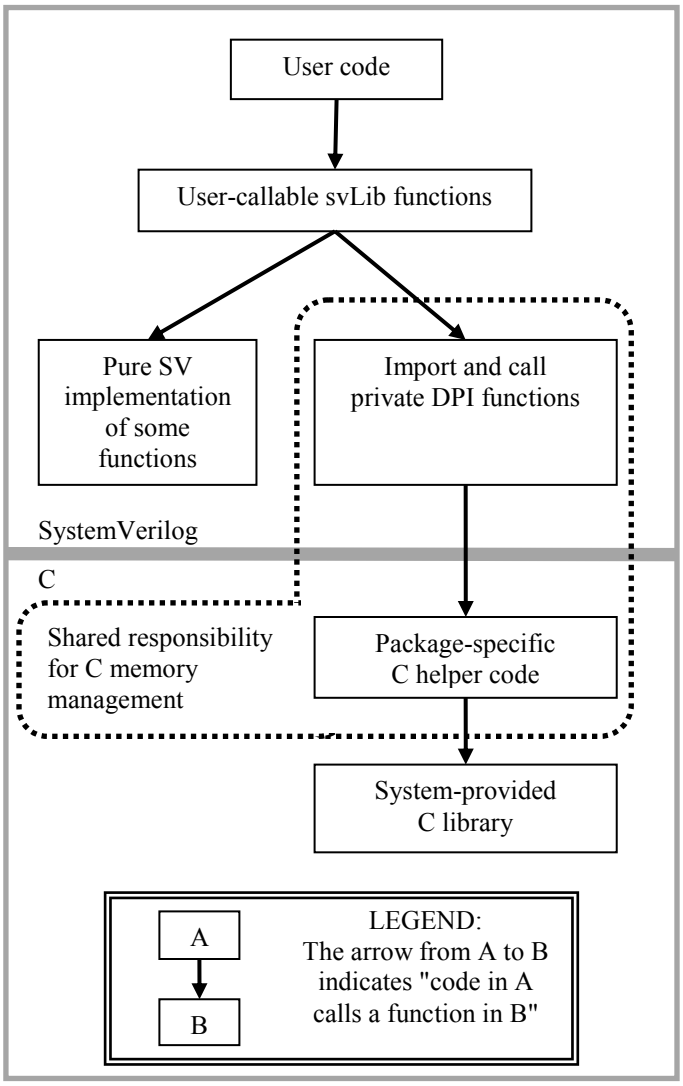

Figure 1: svlib architecture

IV. SPECIFYING THE LIBRARY

We had several design principles in mind from the outset. We found that adhering to these principles when specifying *svlib* was by far the most challenging part of our task. By contrast, the implementation and testing described in section X was comparatively straightforward, and fitted readily into our usual coding practices.

Our core goals, outlined in more detail in later parts of this section, were as follows:

- Convenient API for SystemVerilog programmers, with no C or DPI programming required
- Consistent and uniform handling of error conditions
- Consistent naming conventions
- Portability across simulators, operating systems, and host system architecture
- No dependencies on any other SystemVerilog packages
- Compatibility with all popular verification methodologies
- All deliverables to be open-source, unencumbered by any restrictive license agreements

In addition, we naturally paid attention to ensuring that the library is as free of bugs as possible, does not impose any undue performance penalty, and is well documented. These implementation goals, however, can be revisited and improved in future versions of the *svlib* code base. By contrast, it was essential to meet our core goals from the outset in order to avoid the severe inconvenience to users that might arise if *svlib*'s user-facing specification were to change in the future.

A. Convenient API for SystemVerilog programmers

We have never aimed for *svlib* to offer unique functionality that cannot be found elsewhere. Every operation provided by *svlib* can be found in some other language or library. It is valuable not for any novelty, but by offering useful and powerful facilities in a way that is completely natural and straightforward to use for any SystemVerilog programmer[3]. To make use of the library, no extra effort beyond importing a package should be required of the programmer.

Beyond this simple use model, though, we spent a great deal of effort in considering the most appropriate functions to include and, especially, what form they should take. Two issues gave us particular difficulty: the handling of error conditions as described in subsection IV.B below, and the choice of whether to use an object-oriented approach. The question of whether to present *svlib* as an object-oriented ("OOP" or class-based) package was sufficiently troublesome that we have sidestepped it by providing both an object-oriented API and a straightforward function-call API for many of its features. Users are free to choose whichever approach better suits their purposes and coding style. This issue is discussed more fully in section V.

B. Consistent and uniform handling of error conditions

In any utility library, some of the functions may give rise to error conditions. Sometimes these errors are purely internal (for example, a string overflowing some pre-allocated buffer space), some are genuine usage errors that the programmer probably wants to know about (for example, attempting to discover the modification date of a file that does not exist), and some errors might or might not be considered to be user errors (for example, attempting to examine a file in a directory for which the user does not have read or execute permission). Library users must be given the opportunity to handle any errors gracefully.

1) Error code in an output argument

We briefly considered the possibility of having every function pass its error code through an output argument. However, we quickly dismissed this approach as clumsy and impractical. It forces the user to declare a suitable result variable and explicitly pass it to each function call. Default

[3] There is an important exception to this general aim. From the outset we anticipated that *svlib* would never be appropriate for RTL design. The library makes heavy use of non-synthesizable features such as the DPI, VPI, and dynamic data types. Consequently it is unsuitable for use by RTL designers writing synthesizable code.

output arguments, introduced in the 2009 revision of the SystemVerilog standard, could make this approach tractable. Unfortunately this valuable language feature is not yet implemented in some commercially available simulators at the time of writing.

2) Error code as a function return value

Our first attempt to achieve uniformity was based on having every *svlib* function, without exception, return an integer error code as its result. In most cases there are many possible error conditions, but only one success code is required. Consequently we chose to have such functions return zero (false) on success, and an appropriate integer code for any kind of error.

Consider, for example, the function that reports the current working directory as a string. Using this approach, the function prototype and an example of its use would be:

```
// declaration in svlib_pkg:
function int sys_getCwd(output string dirname);

// user code:
import svlib_pkg::*;
int err;
string cwd;
err = sys_getCwd(cwd);
if (err)
  $display("svlib error %0d occurred", err);
else
  $display("working directory is %s", cwd);
```

Although attractive for its consistency, this approach was very inconvenient in practice. It imposed a significant burden on the programmer to capture every function's result in an error variable or by means of a test, when for many of the library's functions it is more natural simply to return a result value.

However, this function is essentially free of possible error conditions and would be much more natural if expressed thus:

```
// declaration in svlib_pkg:
function string sys_getCwd();

// user code:
import svlib_pkg::*;
string cwd;
cwd = sys_getCwd();
$display("working directory is %s", cwd);
```

In addition, the need to use zero (false) for success made for user code that was error-prone and hard to read, and introduced serious difficulty with the naming of some functions. For example, we have a function `regex_match` that tests a string against a regular expression. But that name was completely inappropriate when the return value was to be zero (false) on success.

Although it was a hard decision to reject a solution that offered such complete consistency, we felt that this approach was just too clumsy to be comfortable, and we abandoned it.

3) Our chosen solution

Following our avowed aim of making the library as painless as possible for SystemVerilog programmers to use, we finally settled on a different approach.

As far as possible, functions are designed so that in the event of user error they return a result that is sensible and – most importantly – not misleading. For those few functions where the user clearly has a responsibility to anticipate and handle an error condition, we provide an output argument that the user must populate with an error-code variable. Other functions are unencumbered by error reporting machinery.

This leaves the problem of how to handle errors that the user would not normally anticipate. In these cases we detect the error in procedural code within the library, but then use a SystemVerilog immediate assertion to throw an error and provide detailed diagnostics at the console. Using an immediate assertion in this way makes it possible for a programmer to suppress the error using `$assertoff` or some equivalent tool command. We also provide package configuration functions that allow users to do the same thing without needing to know the names of the assertions in question.

It is appropriate to observe here that errors detected by our library are invariably reported on the SystemVerilog side. Many such errors in fact originate in C code that was called through the DPI, but such errors are always handled defensively in our C support code and passed back to SystemVerilog for presentation to the programmer in a SystemVerilog-friendly form.

For users who wish to handle errors for themselves, we also provide a means to test for any error that may have occurred in the most recent *svlib* function call. This information is recorded separately for each SystemVerilog process, so that there is no risk of some other process disturbing the error information between occurrence of an error and the user's code testing for it.

The per-process error recording mechanism is described in section VI. It provides a useful compromise for the conflicting demands of robustness, convenience and flexibility:

- By default, library errors cause an assertion-style error message, with the function that caused the error taking defensive default actions and returning some default result.
- Users can easily suppress these error messages if required, accepting the risk of occasional unexpected behavior if the programmer does not test for such errors explicitly.
- Advanced users retain the ability to test explicitly for library errors and handle the consequences in any way they choose.

C. Consistent naming conventions

svlib provides various kinds of named items that a user may need to mention in their code: packages, classes, functions and their arguments, and enumerated types with their enumerator[4] names. Users value consistent naming to make item names not only memorable and somewhat descriptive, but also easy to find in the documentation.

As with many matters of style, this is sure to involve some compromise. Descriptive names are inevitably longer and therefore more cumbersome. For each major group of functionality we have attempted to prepare practical examples of its use. That experience has sometimes led us to change our opinion about what names are most suitable. Generally we have chosen the following guidelines:

- `camelCase` for names that are formed from more than one word, such as `matchStart`;
- underscore separator between a prefix or suffix (identifying a group of related ideas or features) and the remainder of a name, as in `regex_match` and `file_mTime`;
- short names wherever possible for class methods, which will always be prefixed by an object name;
- a compact name for the package itself, so that it remains reasonable to use explicit package prefixes if you wish to avoid wildcard importing of all names in the package;
- long and very descriptive names for internal functions and other items such as DPI imported function names that should never be referenced by users, to minimize any risk of collisions with ordinary user-defined names.

D. Portability across simulators and platforms

From the outset we were committed to ensuring that *svlib* is fully portable across all major simulators. SystemVerilog simulators from major vendors have now reached a high level of maturity and very little tool-specific treatment was required to achieve compatible behavior with the three tools we used.

However, these tools are now widely deployed on a variety of platforms, often but not always using the Linux operating system where the distinction between 32-bit and 64-bit platforms is important. We have tested the current release on a range of platforms but we cannot be sure of having covered all possibilities. Instead, we have taken great care to ensure that the implementation is as robust as possible against such things as endianness or 32/64-bit differences, and we hope to engage the community in helping to extend the set of fully supported platforms.

Microsoft Windows® is also used with some SystemVerilog simulators, but at the time of writing we have been unable to extend our testing to that platform. Differences between the Windows and Linux C libraries may be expected to present some challenges.

[4] The word *enumerator* denotes one of the named values that make up an enumerated type's set of permissible values. The authors prefer the portmanteau word *enumeral*, but it appears to have been hijacked by GCC's documentation as a synonym for "enumerated type".

E. No dependencies on any other SystemVerilog packages

svlib aims to provide services to any SystemVerilog programmer. As such it must be a truly stand-alone library, capable of deployment in any user environment regardless of what other packages that environment uses.

svlib does have some dependencies, of course. It makes heavy use of functions from the standard C library, which we assume to exist on any practical simulation platform. It also makes use of one third-party C package, the `libyaml` open-source YAML parser/writer. A copy of this package is bundled with the distribution. We believe that neither of these dependencies will be burdensome for the SystemVerilog programmer.

Although there is always a possibility of name collisions between a new package and users' existing code, such collisions can always be sidestepped if necessary by using fully qualified package names, or explicit import of individual names, rather than wildcard package import.

Finally, we should note that any developers who wish to extend or modify the library will probably wish to make use of the SVUnit unit-testing framework as we did. As with the other dependencies, we anticipate that this will present no significant obstacle.

F. Compatibility with all popular verification methodologies

svlib makes no attempt to trespass on the territory of verification methodology libraries such as the UVM. It aims to be complementary to them, providing functionality that will be as useful in a UVM-based testbench as in any other verification code. It can be used alongside any published or proprietary verification methodologies with no special considerations required. As described in section IV.B, *svlib*'s built-in error messages can easily be suppressed, allowing the programmer to detect error conditions and report them through a methodology-specific reporting mechanism if desired.

Inevitably there is some overlap of features between *svlib* and verification base class libraries such as the UVM. For example, the UVM already provides regular expression matching and interrogation of the simulator command line (although in both these cases *svlib*'s support is considerably more comprehensive and flexible). Although there are no fundamental problems caused by this overlap, we have attempted to choose names for any such functions in a way that minimizes any confusion with comparable UVM features.

There is, however, one specific area where there is an interesting possible synergy with the UVM. As discussed in section VIII, we offer convenience macros to automate the serialization and deserialization of user-specified objects to and from a configuration file holding a textual representation of such objects. These macros have a very similar look and feel to the field automation macros of the UVM. If, at some future time, *svlib* were to be distributed with the UVM as a matter of course, the UVM field automation macros could easily be augmented to provide this serialization support, rendering our serialization macros unnecessary in a UVM environment.

Similar considerations apply to the OVM[8], which has close similarities to the UVM and is still in active use.

G. *All deliverables to be open-source*

svlib has initially been developed by a small group within our company. Its ambition, though, is to offer a utility package that will be widely adopted among the SystemVerilog user base. This goal can be met only if it is fully open-source. We have applied the rather liberal Apache open-source license [9] to it, just as Accellera has done with the UVM. It is hoped that this will present no obstacles either to adoption or to community involvement with its development.

We use the third-party `libyaml` package. This software is available under the MIT open source license [10], which is similarly permissive.

V. Class Methods or Package Functions?

Our initial instinct was to present *svlib* to the user as a set of SystemVerilog classes. We imagined that we would have a class to represent a regular expression, another class to represent a filename, and so on. From the library writer's point of view, packaging user data in this way is extremely attractive because it allows us to associate arbitrary hidden data with each object. In particular, this hidden[5] data could allow our SystemVerilog code to keep track of what was happening in the C world, allowing many optimizations because we could often avoid passing information across the DPI boundary thanks to this local knowledge of the C state.

As soon as we began to sketch out the user-facing API, though, we immediately encountered a serious problem of usability. SystemVerilog users will typically expect library operations to accept input arguments that are native SystemVerilog types (particularly strings), process them in various ways, and return their results also as native data types. A good example is provided by the string `trim` function, which removes leading and/or trailing whitespace from a string. There are at least four plausible ways to provide this functionality:

- as a simple function that accepts a string argument and returns a string result;
- as an operation on an object representing the string, changing the object's contents to represent the trimmed string;
- as an operation on an object representing the string, leaving the object's original contents untouched, and returning a new object containing the trimmed string;
- as an operation on an object representing the string, leaving the object's original contents untouched, and returning a SystemVerilog string containing the trimmed value.

[5] Inevitably, in an open-source package nothing is truly hidden. `protected` class members are visible but inaccessible to user code. Naming conventions and documentation are used to indicate other parts of the library, such as the "private" package `svlib_private_base_pkg`, that should never be accessed directly by user code but for which the language itself cannot enforce access restrictions.

Of those four approaches, only the first is likely to be attractive to SystemVerilog users who have become familiar with the behavior of the language's native string data type. But the advantages to the library writer of an object-based representation are too great to ignore, especially for functions that must do significant work on the C side of the DPI.

We finally decided to adopt a hybrid approach in which classes, defined by *svlib*, are used to represent almost all user data. The methods of these classes are available for any user who prefers the explicit class-based representation. However, we anticipate that many users will prefer a straightforward function-call interface at the package level. We have therefore implemented the classes with more concern for efficiency and ease of library management than for usability, and we have provided a set of package-level convenience functions around those classes. In this way we hope to get the best of both worlds, with a robust and extensible class-based infrastructure that nevertheless can be used in a very straightforward way through the convenience functions for users who prefer not to adopt the OOP approach. Careful implementation of the convenience functions minimizes the amount of data copying performed.

An interesting consequence of this decision is that each user-facing function requires one or more new objects to be brought into use. If this were done in the obvious way by constructing the objects on demand, it would probably impose an unreasonable burden on SystemVerilog's memory allocation and garbage collection. To mitigate this, the library maintains a pool of recyclable objects that can be used internally by the convenience functions. These objects are used only briefly by each call to a convenience function, and are then returned to the pool for use by future function calls.

To illustrate the way this model is presented to users, here are some excerpts from the declarations of the `Str` class (which is a wrapper for a string) and its `trim` function, together with examples of how a user might apply them.

```
package svlib_pkg;
 // svlib object that holds a string
 class Str extends svlib_base;
  typedef enum {NONE,LEFT,RIGHT,BOTH} side_enum;
  extern static function Str
      create(string contents="");
  extern function string get();
  extern function void trim(side_enum side=BOTH);
  ...
 endclass
 // Convenience wrapper for the trim function
 function string str_trim(
           string s, Str::side_enum
side=Str::BOTH);
 ...
```

The `create` method of class `Str` acts as a constructor, but has some useful additional properties that are discussed in more detail in section VII. Users are discouraged from directly calling the constructor of any *svlib* class.

The `get` method, not surprisingly, returns the object's string contents as a native SystemVerilog string.

It is noteworthy that we have chosen to put the definition of enumeration type `side_enum` into the `Str` class, rather than exposing it at package level. If it were a package-level definition, the short names `NONE`, `LEFT`, `RIGHT`, `BOTH` would be injected into the user's namespace by a wildcard import of the package. This would be very likely to lead to name collisions. By concealing its typedef in a class, we keep the names short and descriptive while enforcing the use of a simple and mnemonic `Str::` prefix when they are used.

Equipped with this set of declarations, we can now perform string trimming in two different ways. A user who has a simple one-off need to remove surplus space from the beginning or end of a string can use the convenience function directly:

```
import svlib_pkg::*;
...
string s = "  needs trimming    ";
$display("=%s=", str_trim(s, Str::RIGHT));
    // displays "=  needs trimming="
```

By contrast, a user who needs to perform many successive operations on a string, or who wishes to be able to pass string objects around easily by reference, may find it preferable to use the `Str` object representation:

```
import svlib_pkg::Str;
...
Str s = Str::create("  needs trimming    ");
s.trim(Str::RIGHT);  // do the trim operation
... // possibly do further operations on 's'
$display("=%s=", s.get());
    // displays "=  needs trimming="
```

VI. Per-Process Error Handling

As described earlier, we chose to report errors by capturing error information from the most recently-failed library function call, making this available to users through a query function (which cannot itself yield any errors, and does not affect the library error status). To avoid any risk of corrupting the error information in one process by the activity of some other process, we record this error information independently for each process.

This is straightforward to implement by maintaining an associative array of error objects indexed by process handle. Unfortunately, one widely used commercial SystemVerilog simulator does not support associative arrays indexed by class or process handle. Fortunately this same tool (alone among currently available simulators) provides a unique string name for any process, obtained by formatting the process handle itself as a string using the `%p` formatter. For that simulator only, we keep per-process information in an associative array with string index type. This imposes a slight performance penalty, measured as about 1.3µs of runtime per library function call on a modest 32-bit Linux workstation. Although this performance impact is unfortunate, we believe it to be an acceptable price to pay for the convenience and robustness of per-process error tracking.

VII. Effects on Random Stability

In SystemVerilog, the creation of any new object of class type disturbs the random number seed of the creating process, as described in clause 18.14 of [6]. This is problematic for our library because, although *svlib* itself does not participate in random number generation in any way, it creates objects on the fly in a way that is very hard for users to predict or control. Since *svlib* is much concerned with string and file processing, it seems likely that it will be heavily used in diagnostics that may be inserted into or removed from the user's codebase as development and debugging proceeds. If the introduction of debug code affects random stability, the debugging task becomes much more difficult because it is impossible to reproduce a problem case after instrumentation code is added.

To mitigate this problem, we have implemented the library in a way that guarantees it will have no impact on random stability, regardless of how many objects it creates. This is achieved by outlawing use of the constructor of any *svlib* class, instead delegating construction to a static `create` method provided by each class. The relevant parts of the `create` method of some imaginary class `svlib_C` are shown here.

```
class svlib_C extends svlib_base;
  static function svlib_C create(...);
    std::process p = std::process::self();
    string rs = p.get_randstate();
    create = new(...);
    p.set_randstate(rs);
    ... // any other initialization activity here
  endfunction
  ...
```

By saving and restoring the randomization state of the calling process across object construction, we guarantee that *svlib* cannot disturb the user's randomization and therefore users are free to add or remove *svlib*-based debugging code with confidence.

A compile-time macro can be defined to defeat this mechanism, removing its (rather small) performance overhead but sacrificing the guarantee of random stability during debug. This option is intended for confident users who have a fully debugged environment that they wish to use for a regression, with many different random seeds, where the highest possible performance is considered to be more valuable than random stability.

VIII. Working with Configuration Files

As hinted in the introduction, an important driver for the development of *svlib* was our desire to be able to read and write configuration data files in structured formats such as YAML. In our experience it is common for such files to be created, typically by scripts, to reflect the desired DUT and testbench setup for a given simulation run. The verification

environment should be able to read such a file and use its contents to populate one or more configuration data objects.

At first glance this requirement seems to be centered on the problem of reading and writing the desired file format. However, interpreting the file is not sufficient. It is also necessary to provide some automation of the conversion between native data structures and a representation that maps easily onto the file.

As an example, consider these SystemVerilog classes, and objects of those classes with some values stored in their data members. Note the hierarchical instantiation of a `simpleConfig` object within a `largerConfig`.

```
class simpleConfig;
  int scalarInt;
  string scalarString;
endclass

class largerConfig;
  int scalarInt;
  simpleConfig objectSC;
endclass

...

simpleConfig sc = new;
sc.scalarInt = 1234;
sc.scalarString = "text in sc";
largerConfig lc = new;
lc.scalarInt = 5678;
lc.objectSC = sc;
```

If we were to write the `largerConfig` object out to a YAML file, we would expect the file's contents to appear approximately as follows:[6]

```
%YAML 1.1
---
# Contents of largerConfig object
scalarInt: 5678
objectSC:
  # contents of simpleConfig object
  scalarInt: 1234
  scalarString: text in sc
```

Although the structure of the YAML data is clear enough, there is no straightforward way to map from it to the contents of our two SystemVerilog objects. For more complex objects, the problem would be even more troublesome. Furthermore, it is likely that not all data members of an object should be represented in an external configuration file. For all these reasons, it became clear that we needed an intermediate representation. Borrowing from other software environments, we chose to think of this representation as a *Document Object Model* (DOM).

A. *Characteristics of the svlib DOM*

In *svlib* a DOM is a tree structure having a single root node. A node in the tree can have one of three basic forms:

- a *map*, whose value is an un-ordered collection of nodes, each node identified by a string name (key)
- a *sequence*, whose value is an ordered list of nodes with each node identified by its position in the list
- a *scalar*, whose value is a "scalar value" object

Clearly, scalars are the leaves of the node tree. We chose to have each scalar's value be represented by a further object in order to make it easy to add new scalar data types in future versions. Currently only integral and string scalars are supported; we have yet to encounter a situation in which this is truly inadequate, but further data types are planned for increased flexibility and convenience. Scalar value classes are parameterized by the type of raw data that they hold, in much the same way that the UVM's resource database uses parameterized SystemVerilog classes to provide resource storage for any user-defined data type. A tutorial introduction to the UVM resource database can be found at [11].

The two structured node types – sequence and map – are sufficient to represent almost any reasonable data structure, but they also map naturally on to native SystemVerilog types. A DOM map is simply an associative array of nodes indexed by string; a DOM sequence is a queue of nodes. In this way, we were able to create a DOM in SystemVerilog with only a few new classes, all derived from two base types:

- `cfgNode` and its subclasses `cfgNodeMap`, `cfgNodeSequence`, `cfgNodeScalar` represent the different kinds of DOM node. A `cfgNodeScalar` contains a `cfgScalar` object representing the scalar value at the node.
- `cfgScalar` and its subclasses such as `cfgScalarString` are containers for scalar values.

Users of the library typically are not expected to interact directly with a DOM. Instead, they will use built-in *svlib* functionality to copy a DOM to or from any of the supported file types (currently YAML and the more restricted INI file format). Similarly, the library's classes offer methods that can access and manipulate individual nodes of a DOM.

B. *Representing user-defined data as a DOM, and vice versa*

Although the DOM structure is very flexible and provides a consistent way to represent any required data structure, it is too clumsy for ordinary use. Users of course expect to create their own configuration data objects (typically using classes) in whatever way is most convenient. There remains, therefore, a problem of how to transfer data between the user's classes and a DOM representation. Users could write their own methods to create and unpack a DOM representation from the object, but

[6] The YAML file format is quite flexible and allows many detailed variations in the way data is represented in the text file. The example shown here is legal YAML, but is by no means the only possible representation. Lines beginning with a `#` character are comments and do not influence interpretation of the data.

this is tedious and error-prone, and automation is obviously desirable. Frustratingly, SystemVerilog lacks the introspection and reflection facilities that are needed to make this fully automatic.[7] To handle this problem we took our cue from the UVM and provided field automation macros to achieve the same result. Using this approach, our `largerConfig` object becomes:

```
class largerConfig extends someUserCfgClass;
  int scalarInt;
  simpleConfig objectSC;
  `SVLIB_DOM_UTILS_BEGIN(largerConfig)
    `SVLIB_DOM_FIELD_INT(scalarInt)
    `SVLIB_DOM_FIELD_OBJECT(objectSC)
  `SVLIB_DOM_UTILS_END
endclass
```

The other class `simpleConfig` is handled in a very similar way.

These macros automatically create the code for two new methods of the class:

```
cfgNode toDOM(string name);
void fromDOM(cfgNode dom);
```

`toDOM` constructs and returns a new DOM map node containing a representation of the object's contents, *including those of the lower-level object*. This DOM can then be written to a file of any supported format, simply by passing it to the `serialize` method of an appropriate `cfgFile` object; in the current version of *svlib*, subclasses `cfgFileYAML` and `cfgFileINI` are provided.

`fromDOM` uses the given DOM's contents to populate the object's data members, again descending as necessary into the inner `objectSC` object.

The following example shows how a `largerConfig` object can be populated from a source file `src.yaml`, modified, and then written out to another file `dst.ini` in a different file format:

```
largerConfig cfg;  // The object to be populated
cfgNode      dom;  // The DOM root node
cfgFileINI   fi;   // INI file reader/writer
cfgFileYAML  fy;   // YAML file reader/writer

// Read the YAML file into a DOM
fy = cfgFileYAML.create();
dom = fy.deserialize("src.yaml");

// Populate cfg from the DOM
cfg = new;        // or perhaps use UVM factory
cfg.fromDOM(dom); // also constructs inner object

// Modify the cfg objects
cfg.scalarInt = 42;
cfg.objectSC.scalarString = "new value";

// Make a new DOM from objects. Name it "NewCfg"
dom = cfg.toDOM("NewCfg");

// Write the new DOM to a .INI file
fi = new;
fi.serialize("dst.ini", dom);
```

Given the YAML file provided earlier, the resulting output in `dst.ini` would be as follows:

```
scalarInt=42

[objectSC]
scalarInt=1234
scalarString=new value
```

We can see that the INI file format uses `[section]` to denote first-level map entries that are themselves maps. Deeper nesting is not supported by INI files; similarly, sequence nodes cannot be written to INI files. YAML and other formats are much more flexible and expressive, and can represent the full range of DOM features.

Full details of these and other features of the DOM and file reader/writer classes can be found in the documentation.

C. *Ambiguities of the DOM*

We have chosen a very lightweight DOM, emphasizing flexibility and simplicity. Consequently there are some aspects of typical user data structures that cannot be fully represented. In particular, map nodes are used for two quite distinct purposes: to represent an associative array indexed by string, and to represent the various named data members of an object. In practice this ambiguity does not present any real difficulty, as the target object should have been designed to match the anticipated data structure. A key advantage of the automation macros is that various syntactic and semantic checks are performed as a DOM is copied to or from an object, so any mismatches will be identified for correction by the user.

D. *Serializable Object as a Common Base Class*

In the UVM, field automation macros create methods such as `copy()` and `compare()` that override virtual methods of the

[7] At first we hoped to use SystemVerilog *attributes*. VPI code would scan the class definition, finding data members with the attribute `(* svLibSerialize *)` and automatically adding them to the DOM. Unfortunately, simulator VPI support for attributes on class members is very patchy and immature, and we were forced reluctantly to abandon this approach.

uvm_object base class. This has the great advantage that you can create infrastructure code, with variables of uvm_object type, that allows you to work on any derived object. In *svlib* the situation is very different. It would be unreasonable and selfish for us to expect users to derive all their classes from some common *svlib* base class. Consequently, the macro-generated toDOM and fromDOM methods are completely separate for each user-defined class that has them, and it is impossible to build common infrastructure to handle serialization in a consistent way for all objects.

The 2012 revision of SystemVerilog, however, provides a solution perfectly matched to this problem: the new *interface class* construct, described in clause 8.26 of [6]. This form of multiple inheritance, inspired by the *interface* features of Java (described in, for example, the tutorial at [12]), allows a user class to *implement*, rather than to inherit from, an interface class that defines a set of virtual methods in much the same way that a virtual base class might do. The user class is then obliged to provide concrete implementations of all methods defined virtual by the interface class. Once this is done, it is legal for a class handle of the interface class type to reference an object of any class that implements that interface class. The following code examples show how this might work in practice.

First we show how *svlib* might define its interface class, specifying the virtual methods that any implementing class is required to override:

```
interface class svlib_serializable;
  pure virtual
    function cfgNodeMap toDOM(string name);
  pure virtual
    function void fromDOM(cfgNode dom);
endclass
```

Next we modify the user's configuration class so that it is defined to implement the interface class. Note that the user's original class hierarchy is not disturbed – largerConfig is derived from someUserCfgClass as before. The implements keyword imposes an obligation on the new class to implement toDOM and fromDOM methods, but that obligation is discharged by the macros so that no additional user code is required:

```
class largerConfig
    extends someUserCfgClass
    implements svlib_serializable;
  int scalarInt;
  simpleConfig objectSC;
  // The macros create toDOM/fromDOM as before,
  // but now these methods override virtual
  // methods of the interface class.
  `SVLIB_DOM_UTILS_BEGIN(largerConfig)
    `SVLIB_DOM_FIELD_INT(scalarInt)
    `SVLIB_DOM_FIELD_OBJECT(objectSC)
  `SVLIB_DOM_UTILS_END
endclass
```

Finally we illustrate an imaginary piece of user infrastructure code that can serialize and deserialize *any* object whose class type implements the svlib_serializable interface class:

```
// This function can take any suitable object
// and serialize it to a YAML file.
function void toYAML(svlib_serializable obj);
  cfgNode dom = obj.toDOM("my_cfg");
  cfgFileYAML fy = new;
  fy.serialize("my_cfg.yaml", dom);
endfunction
```

We have prototyped this approach with very satisfactory results. Unfortunately, one of the popular SystemVerilog simulators does not yet support the interface class feature at the time of writing, and so we felt unable to include this attractive functionality in the initial release.

IX. Simulator Restart, Checkpoint, and Restore

Management of simulator reset/restart, checkpoint, and restore to a saved checkpoint is typically a troublesome problem for the implementer of any VPI/DPI-based package. If the package maintains any state in C memory, that state must be cleared on restart (not especially difficult) and saved and restored across simulator checkpoint/restore (much more difficult). Our solution is to manage all package state in SystemVerilog where it is automatically saved and restored by the simulator's built-in mechanisms.

However, keeping all state on the SystemVerilog side can impose a performance penalty because it may mean that various data objects need to be copied repeatedly across the DPI boundary. To mitigate this cost, we have implemented cached state in C, shadowing the SystemVerilog package state. Because all interaction across the DPI boundary is performed by hidden private functions and not by user-callable code, we can do this with confidence.

On any simulator restart – whether to time-zero reset or to a saved checkpoint – all memory managed by our C code is deallocated, effectively wiping our state cache. Subsequent library function calls will then suffer the overhead of rebuilding the cache, but in every other way the memory management is completely invisible to the user.

Because this memory caching is completely invisible to users, we will be able to use it more and more aggressively over successive implementations without risk of breaking user-visible functionality. Our initial implementation has minimal caching, and consequently has suboptimal performance, but is fully functional and has allowed us to conduct usability tests. In due course, the cache mechanism will be implemented for those functions (notably regular expression processing and the YAML file reader) where it will obviously bring benefits.

X. Implementation, Testing and Documentation

As already noted, the implementation of *svlib* was generally trouble-free. By far the most challenging aspect of its development was to establish a convenient user-facing API that is natural for SystemVerilog programmers to use. The API and

other features were established iteratively over extensive discussions with many colleagues participating.

A. Testing

We aimed to use a test-driven code development methodology. The SVUnit framework [13] is ideally suited to the testing of classes and packages made up of many self-contained functions that can be tested individually, and we used it from the outset. In most cases we were able to construct at least a simple test harness for each new feature before it was developed, and indeed the resulting SVUnit test cases provided a very useful sanity check that we had a sensible API in each case. As an example, here is a fragment of our unit test for the string "find substring" functions `first` and `last`.

```
string sought;
my_Str.set("012345678901234567890");
// Seek something that's not in the string
sought = "A";
`FAIL_UNLESS_EQUAL(my_Str.first(sought), -1)
sought = "1235";
`FAIL_UNLESS_EQUAL(my_Str.first(sought), -1)

// Find occurrences of a single character
sought = "0";
`FAIL_UNLESS_EQUAL(my_Str.first(sought),      0);
`FAIL_UNLESS_EQUAL(my_Str.first(sought, 1),  10);
`FAIL_UNLESS_EQUAL(my_Str.first(sought, 11), 20);
`FAIL_UNLESS_EQUAL(my_Str.last(sought),      20);
`FAIL_UNLESS_EQUAL(my_Str.last(sought, 1),   10);

// Find occurrences of a longer string
sought = "0123";
`FAIL_UNLESS_EQUAL(my_Str.first(sought),  0);
`FAIL_UNLESS_EQUAL(my_Str.last (sought), 10);
```

These methods have the prototype

```
int first(string sought, int ignore=0)
```

where `sought` is the string to search for, and `ignore` is the number of characters at the start of the string (or at the end, for `last()`) that should be skipped-over by the search. Their result is the character position, in the original string, of the leftmost character of the sought substring.

1) Rapid feedback on implementation success

This approach gave us immediate feedback on trivial implementation bugs that might otherwise have been hard to locate. Each unit test runs very rapidly, giving a success/failure indication on any code change in less than twenty seconds even when run on three popular simulators. With this ability to test every new change on all supported platforms in less time than it takes to get a cup of coffee, we were highly motivated to keep the codebase working correctly at all times. Implementation errors were quickly pinned down to a single test case by SVUnit's simple and clear reporting mechanism.

2) Continuous sanity checking of the library API

Unit testing also provided us with an early indication of whether the user-facing API was convenient and sensible, because we were obliged to write test code using that API before it was implemented. Poor design decisions were quickly and effectively highlighted. For example, one of our early attempts had some functions with more than one optional argument (*i.e.* argument having a default value). It quickly became clear that this was extremely error-prone for users, as it was very easy to supply a value to the wrong optional argument and get surprising behavior.

3) Unit tests as a documentation pool

Finally, the ever-growing suite of tests continues to provide a valuable collection of simple usage examples for the various functions, with the benefit that the result of each example is clearly documented by the test itself.

B. Documentation

It has become popular to use automatic document generators such as Doxygen [14] or NaturalDocs [15] for the documentation of packages of this kind. They have the obvious and important benefit that the documentation is automatically kept up to date with respect to changes in the code. However, we have chosen to buck this trend and revert to traditional manually generated documentation. This seemingly retrogressive step was taken only after careful thought. Documentation generated from code and its comments tends to focus exclusively on the properties of individual code fragments down to the function level, whereas we believe a human reader is more likely to be interested in the overall rationale for the package, the relationships among its components, and any guiding principles that it follows as a whole. This information we believe is better captured in human-written material.

The overwhelming advantage of auto-generated documentation is found when the code base is changing rapidly, and complex relationships among (for example) classes in a hierarchy must be kept under review. For a package such as *svlib* whose user-facing behavior should be very stable, this advantage is much less telling.

XI. Performance

Although *svlib* provides convenient access to functionality that previously was hard to reach from SystemVerilog, that convenience would be useless if its performance was unacceptably slow. We have measured the performance cost of selected *svlib* features and we believe that it is acceptable, because performance-hungry operations such as file access and manipulation of large strings are likely to be used only infrequently.

As a convenient benchmark we tried a search-and-replace in which a 2000-character string was scanned using a regular expression that found 100 instances of a 10-character pattern, performing a replacement on each instance. The runtime cost of this operation, on a modest 64-bit Linux machine, was between 5 and 12 milliseconds for three different simulators (with default optimization in force in each case). This appears to be about a factor of 10 slower than the same operation using scripting languages such as Tcl or Perl. Approximately half of the time was attributable to the regular expression compilation

and execution in the C library, and about half to the string substitution which is performed in native SystemVerilog.

In a second test we timed lookup of a file's modification time, as an example of an operation where most of the work is done by the C library and library overhead should be expected to be insignificant. This indeed proved to be the case, *svlib* being competitive with other languages on this operation.

It is clear from our measurements that there are many performance optimizations that could be applied to string manipulations, particularly regular expressions, and we plan to investigate these opportunities in more detail when other aspects of *svlib* have matured.

XII. Future Work

We have an extensive "shopping list" of future work that we would like to add as resources allow, and if there appears to be any demand for them. Among them we should mention:

- Full support for the Microsoft Windows® platform
- Support for Comma Separated Value (CSV) tabular data files
- Support for SQL access to a database
- Some way to associate an already-open C file handle with a Verilog file identifier. With this mechanism in place, a wide range of interesting new functionality would be available including support for temporary files, and communication through TCP/IP sockets and other network mechanisms.
- A huge range of exciting possibilities opens up if it is possible to establish a tight link between SystemVerilog and a scripting language such as Python. This idea has already been explored in a slightly different way by the PyHVL project [16] and we have hardly begun to consider the possible interactions that are possible in this area.

XIII. Conclusions

An initial release is available for download at the time of publication, and can be found by following the `svlib` link at `www.verilab.com/resources`.

We regard this as a beta-quality release: it has a full feature set, and we have made considerable use of it ourselves but it has not yet been stressed by heavy use in project work and therefore may lack maturity. At this time we welcome feedback from potential users, and we hope to use DVCon as an opportunity to gauge user appetite for such a library.

If this initiative is received enthusiastically, we will begin to seek ways to engage the wider community in supporting it and carrying it forward. The authors welcome any feedback, both on the technical content of svlib and its usefulness in practice.

Acknowledgments

The authors wish to thank the following for their contributions and support.

- Our employer Verilab consistently provides opportunity, support and encouragement for the development of ideas of this kind.
- Cliff Cummings of Sunburst Design, Inc. provided supportive and thorough reviews.
- We have many Verilab colleagues whose constructive reviews, practical support, deep expertise, high expectations, and relentless teasing are a source of inspiration and delight. Special thanks are due to Paul Marriott for his careful final code scrub.
- Dr Gordon McGregor is a former colleague now at Nitero Inc. Gordon's commitment to the highest standards of quality and good sense in software development, his impatience with some of SystemVerilog's shortcomings, and his wise and insightful advice, were instrumental in giving *svlib* its current shape.
- The libyaml package[17], written by Kirill Simonov, saved us many days of implementation work.
- Bryan Morris of Verilab and Neil Johnson of XtremeEda Inc wrote and maintain the SVUnit testing framework, which encouraged us to adopt a test-driven development approach that greatly eased our implementation effort.

REFERENCES

[1] Accellera Systems Initiative, "UVM (Universal Verification Methodology)," [Online]. Available: http://www.accellera.org/downloads/standards/uvm.

[2] D. Crockford, "JavaScript Object Notation," [Online]. Available: http://tools.ietf.org/html/rfc4627.

[3] C. C. Evans, "The Official YAML Web Site," [Online]. Available: http://www.yaml.org/.

[4] P. Hazel, "PCRE: Perl Compatible Regular Expressions," [Online]. Available: http://pcre.sourceforge.net/.

[5] Apache Software Foundation, "Class HierarchicalINIConfiguration," [Online]. Available: http://commons.apache.org/proper/commons-configuration/apidocs/org/apache/commons/configuration/HierarchicalINIConfiguration.html.

[6] IEEE, Standard 1800-2012 for SystemVerilog Hardware Design and Verification Language, New York, NJ: IEEE, 2012.

[7] J. Bromley, "If SystemVerilog Is So Good, Why Do We Need the UVM?," in *FDL*, Paris, 2013.

[8] Mentor Graphics, Inc., "OVM and UVM," [Online]. Available: https://verificationacademy.com/topics/verification-methodology.

[9] Apache Software Foundation, "Apache License, version 2.0," 2004. [Online]. Available: http://www.apache.org/licenses/LICENSE-2.0.

[10] Massachusetts Institute of Technology, "The MIT License," [Online]. Available: http://opensource.org/licenses/MIT.

[11] V. Cooper and P. Marriott, "Demystifying the UVM Configuration Database," in *DVCon*, San Jose, CA, 2014.

[12] Oracle, Inc, "Lesson: Interfaces and Inheritance," [Online]. Available: http://docs.oracle.com/javase/tutorial/java/IandI/index.html.

[13] N. Johnson and B. Morris, "AgileSoC SVUnit," [Online]. Available: http://www.agilesoc.com/open-source-projects/svunit/.

[14] D. van Heesch, "Doxygen," [Online]. Available: http://www.stack.nl/~dimitri/doxygen/index.html.

[15] G. Valure, "NaturalDocs," [Online]. Available: http://www.naturaldocs.org.

[16] T. Loftus, J. Greene and B. Smith, "PyHVL," [Online]. Available: http://pyhvl.sourceforge.net/.

[17] K. Simonov, "LibYAML," [Online]. Available: http://pyyaml.org/wiki/LibYAML.