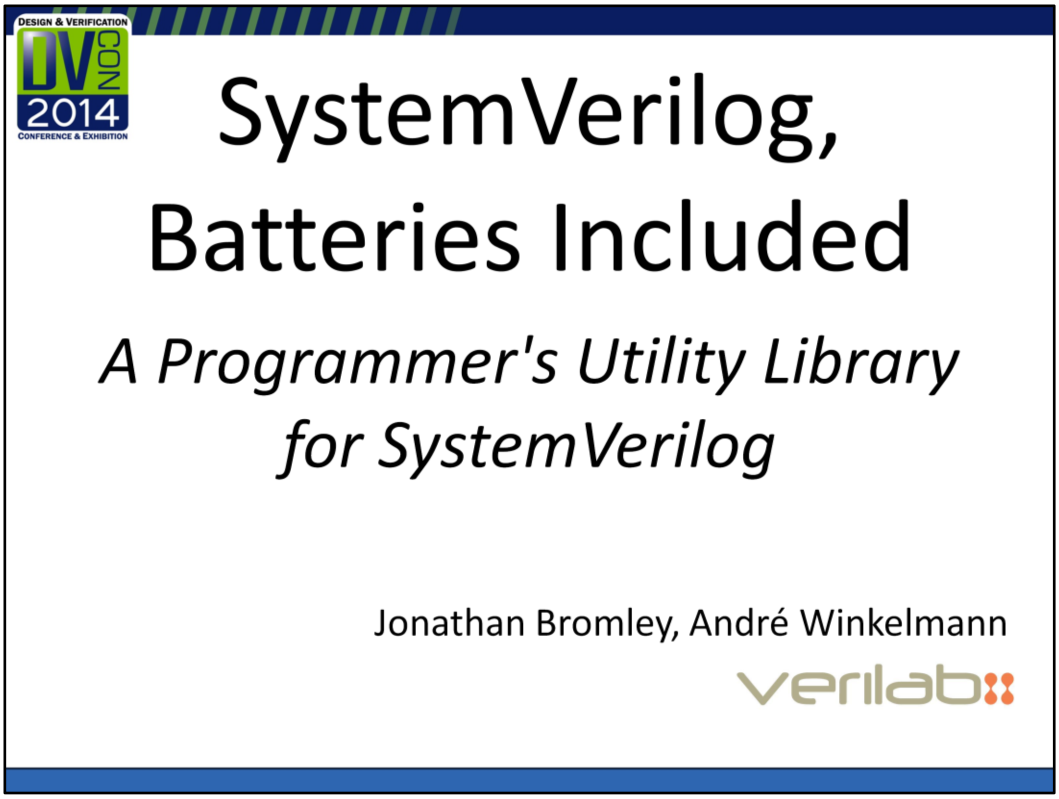
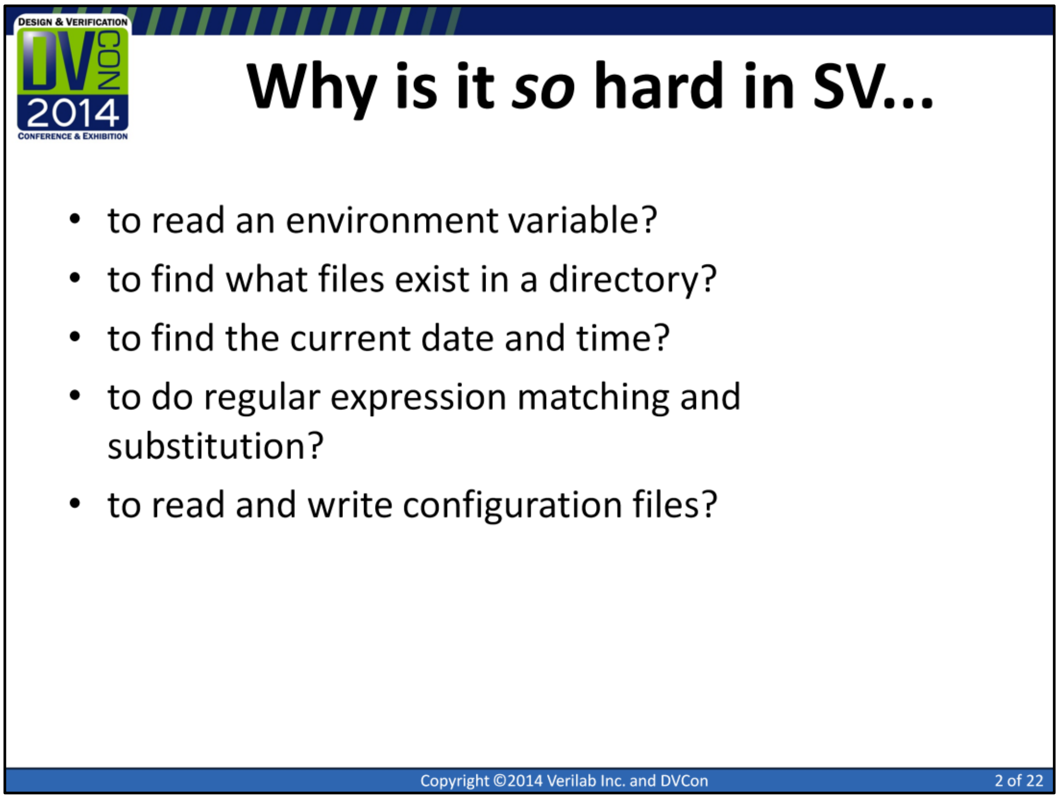


SystemVerilog, Batteries Included

A Programmer's Utility Library for SystemVerilog

Jonathan Bromley, André Winkelmann

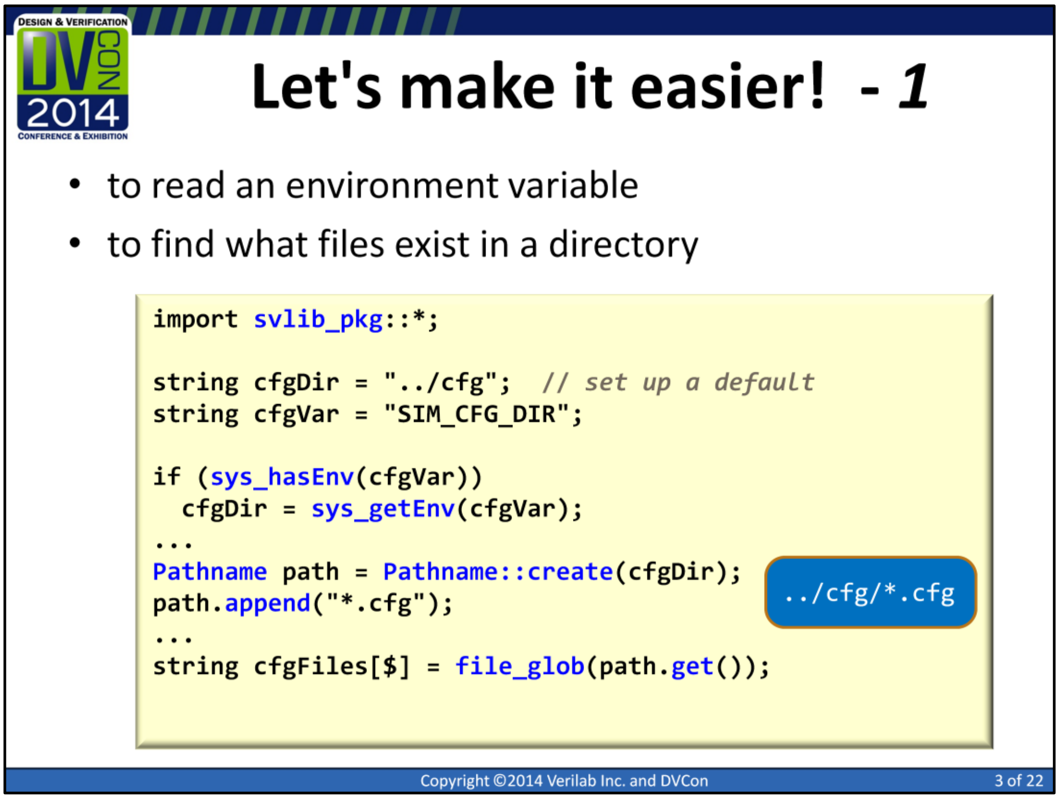
This paper was presented at DVCon in San Jose, CA on 4th March 2014.



Why is it *so* hard in SV...

- to read an environment variable?
- to find what files exist in a directory?
- to find the current date and time?
- to do regular expression matching and substitution?
- to read and write configuration files?

We were motivated to develiop the library described in this presentation because we were frustrated by the lack of features in SystemVerilog that we would take for grarnted in almost any other programming environment.

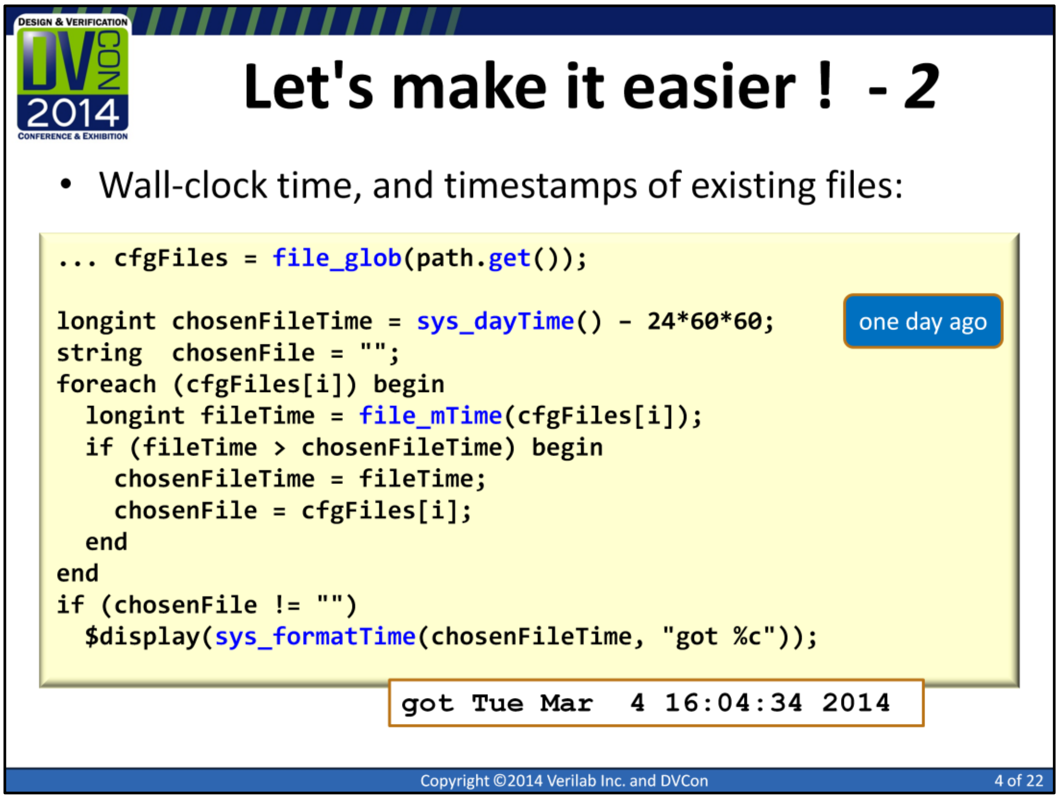


Let's make it easier! - *1*

- to read an environment variable
- to find what files exist in a directory

```
import svlib_pkg::*;

string cfgDir = "../cfg";  // set up a default
string cfgVar = "SIM_CFG_DIR";

if (sys_hasEnv(cfgVar))
  cfgDir = sys_getEnv(cfgVar);
...
Pathname path = Pathname::create(cfgDir);
path.append("*.cfg");
...
string cfgFiles[$] = file_glob(path.get());
```

Our open source library, svlib, helps to ease this problem by providing a range of utility features. Here is an example of svlib being used to:
•find the value of an operating system environment variable using the sys_hasEnv and sys_getEnv functions,
•manipulate a fiile pathame in a convenient way using the svlib Pathname class;
•get a list of files in a directory using the file_glob routine



Let's make it easier ! - 2

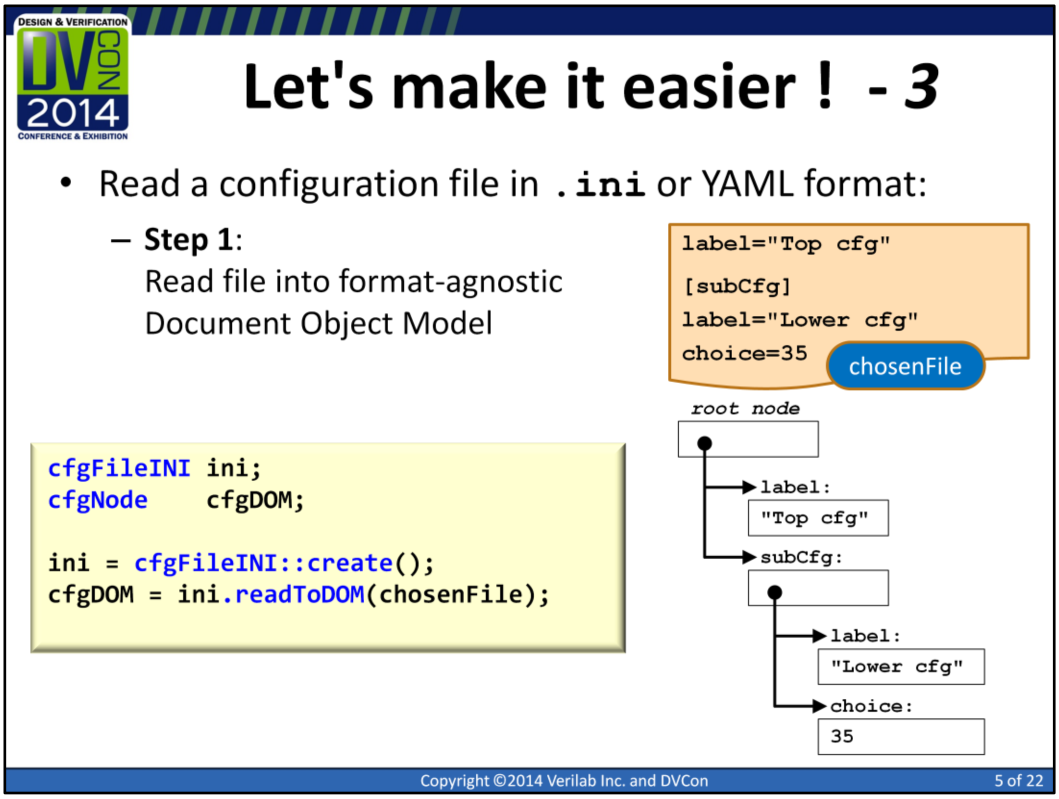
- Wall-clock time, and timestamps of existing files:

```
... cfgFiles = file_glob(path.get());

longint chosenFileTime = sys_dayTime() - 24*60*60;
string  chosenFile = "";
foreach (cfgFiles[i]) begin
  longint fileTime = file_mTime(cfgFiles[i]);
  if (fileTime > chosenFileTime) begin
    chosenFileTime = fileTime;
    chosenFile = cfgFiles[i];
  end
end
if (chosenFile != "")
  $display(sys_formatTime(chosenFileTime, "got %c"));
```

one day ago

```
got Tue Mar  4 16:04:34 2014
```

Having obtained a list of files, we can scan them to determine which is the most recent. We also wish to check that the most recent file is no more than one day old. That requires us to find the time of day using the sys_dayTime function. The modification date (timestamp) of any file can be determined using the file_mTime function. Finally, we can render a time (which is reported in the Unix seconds-since-1970 format) as a human-readable date/time using the sys_formatTime function.



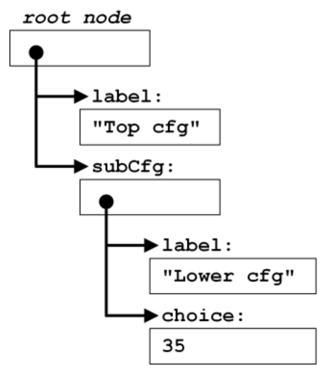
Let's make it easier ! - *3*

- Read a configuration file in `.ini` or YAML format:
  - **Step 1**:
    Read file into format-agnostic Document Object Model

```
label="Top cfg"
[subCfg]
label="Lower cfg"
choice=35
```

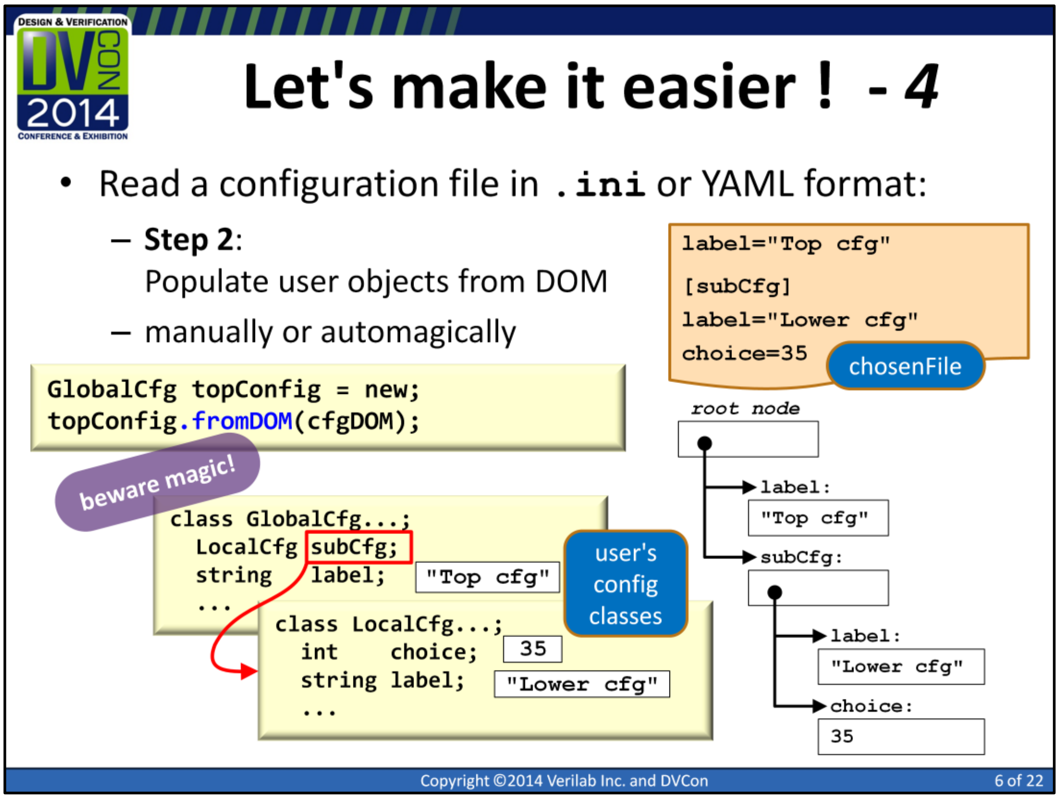
chosenFile

```
cfgFileINI ini;
cfgNode    cfgDOM;

ini = cfgFileINI::create();
cfgDOM = ini.readToDOM(chosenFile);
```



A major feature of svlib is its ability to read and write configuration files in .ini or YAML formats. To make it as easy as possible to add support for new file formats in the future, we use a format-independent Document Object Model representation, and the first step in reading a configuration file is to read the file into this representation using the readToDOM function of the appropriate file class. There is also a writeFromDOM function so that a DOM can be written back to a file.

NOTE: Unfortunately it was not possible to provide YAML support in the first version of svlib. Support for this file format will be added later in 2014.



Let's make it easier ! - 4

- Read a configuration file in `.ini` or YAML format:
  - **Step 2**:
    Populate user objects from DOM
  - manually or automagically

```
label="Top cfg"
[subCfg]
label="Lower cfg"
choice=35
```

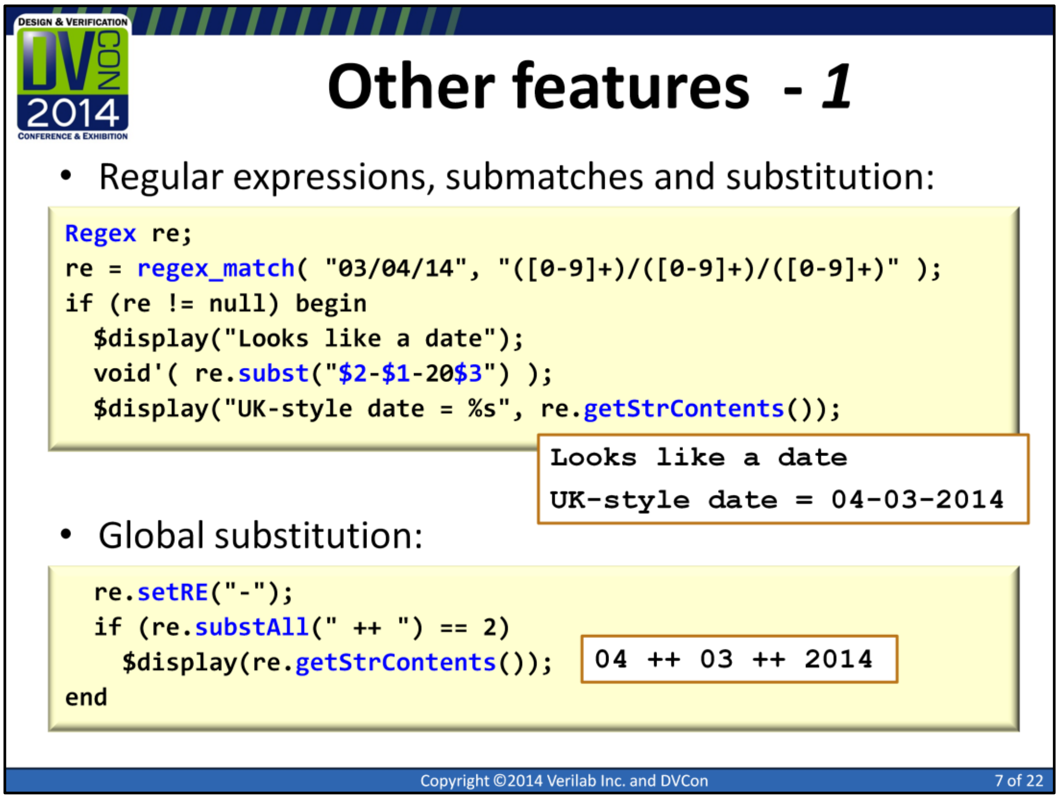
```
GlobalCfg topConfig = new;
topConfig.fromDOM(cfgDOM);
```

```
class GlobalCfg...;
  LocalCfg subCfg;
  string   label;
  ...
```

```
class LocalCfg...;
  int    choice;
  string label;
  ...
```

chosenFile
beware magic!
user's config classes
"Top cfg"
35
"Lower cfg"
root node
label:
"Top cfg"
subCfg:
label:
"Lower cfg"
choice:
35

Once we have the DOM representation, we can use it to populate a user-defined SystemVerilog object. There is some non-obvious automation in play in this example – we`ll describe it in more detail later in the presentation – but the example really does work as shown.



Other features - *1*

- Regular expressions, submatches and substitution:

```
Regex re;
re = regex_match( "03/04/14", "([0-9]+)/([0-9]+)/([0-9]+)" );
if (re != null) begin
  $display("Looks like a date");
  void'( re.subst("$2-$1-20$3") );
  $display("UK-style date = %s", re.getStrContents());
```

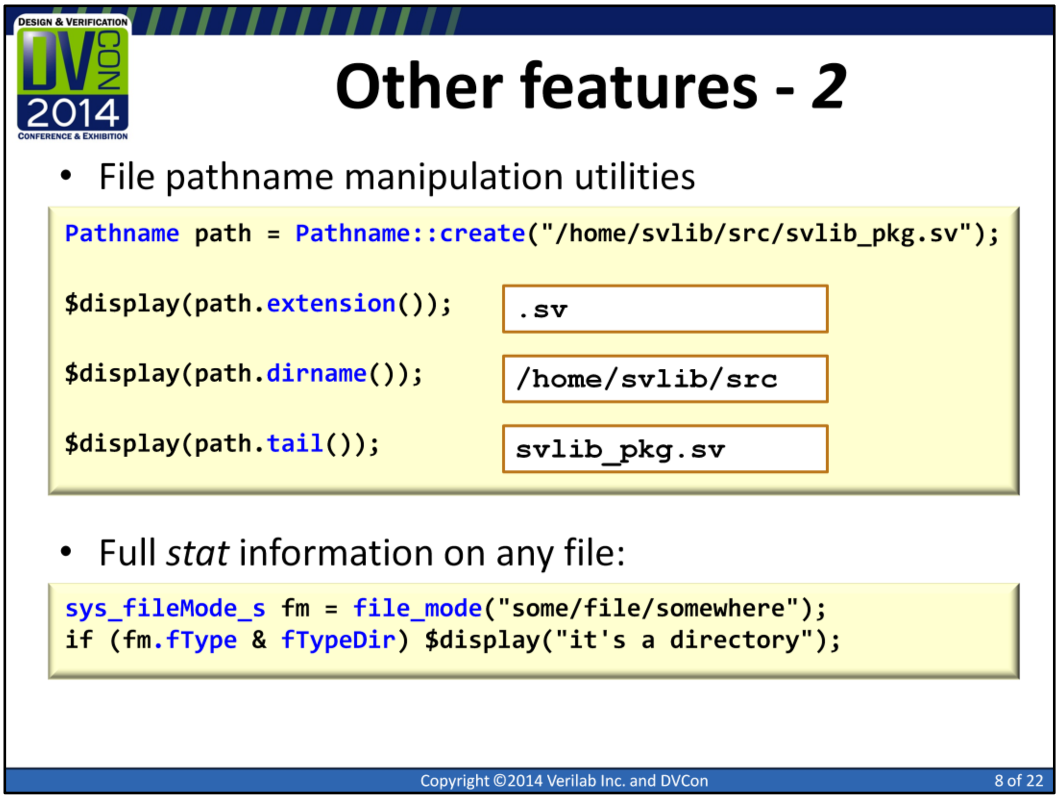
```
Looks like a date
UK-style date = 04-03-2014
```

- Global substitution:

```
  re.setRE("-");
  if (re.substAll(" ++ ") == 2)
    $display(re.getStrContents());
end
```

```
04 ++ 03 ++ 2014
```

We have provided full-featured regular expression matching and substitution as part of svlib. It usese the POSIX extended regular expression library. You can find full details of how to use it in the svlib documentation.



Other features - 2

- File pathname manipulation utilities

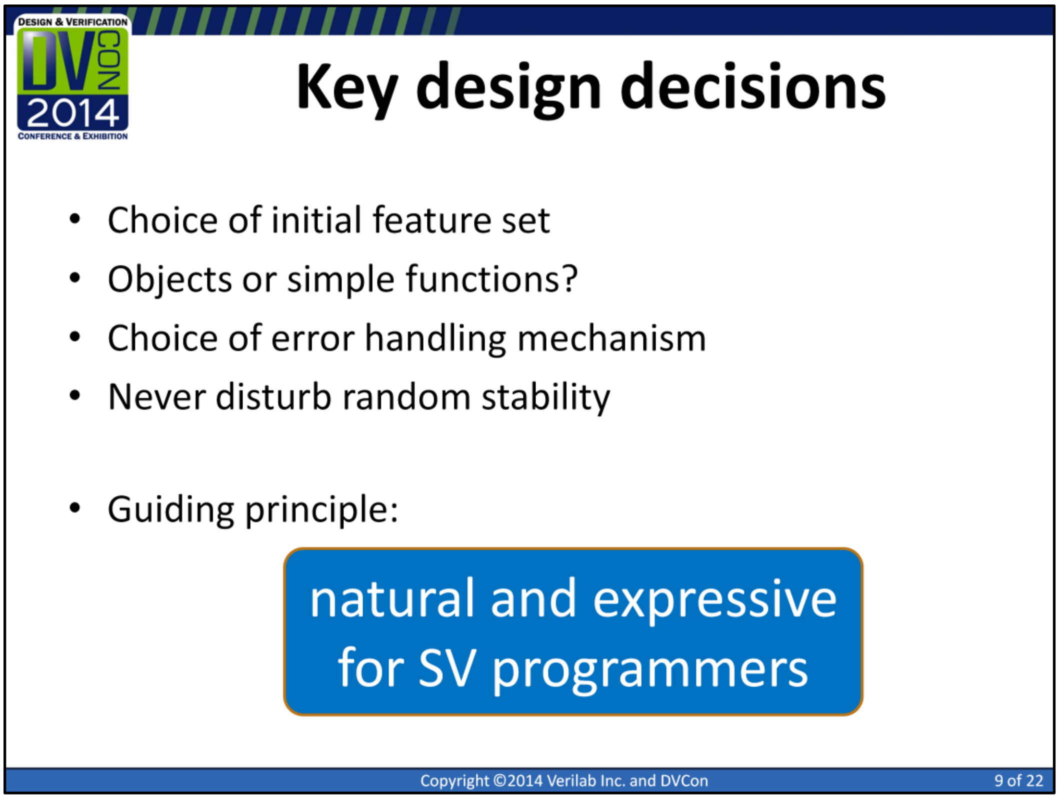
```
Pathname path = Pathname::create("/home/svlib/src/svlib_pkg.sv");

$display(path.extension());        .sv

$display(path.dirname());          /home/svlib/src

$display(path.tail());             svlib_pkg.sv
```

- Full *stat* information on any file:

```
sys_fileMode_s fm = file_mode("some/file/somewhere");
if (fm.fType & fTypeDir) $display("it's a directory");
```

The first box on this slide shows a few more examples of the facilities provided by the Pathname class. It provides manipulation of filenames, carefully preserving the correct number of directory separator characters and being aware of the difference between absolute and relative paths.

The second box shows how more detailed file system queries can be performed. The file_mode function returns a struct that can be examined to find various properties of the file such as ownership, whether it`s a directory or symbolic link, and so on. Other functions allow you to check whether the simulation has permission to read, write or execute a given file, and even to query whether a file exists.



Key design decisions

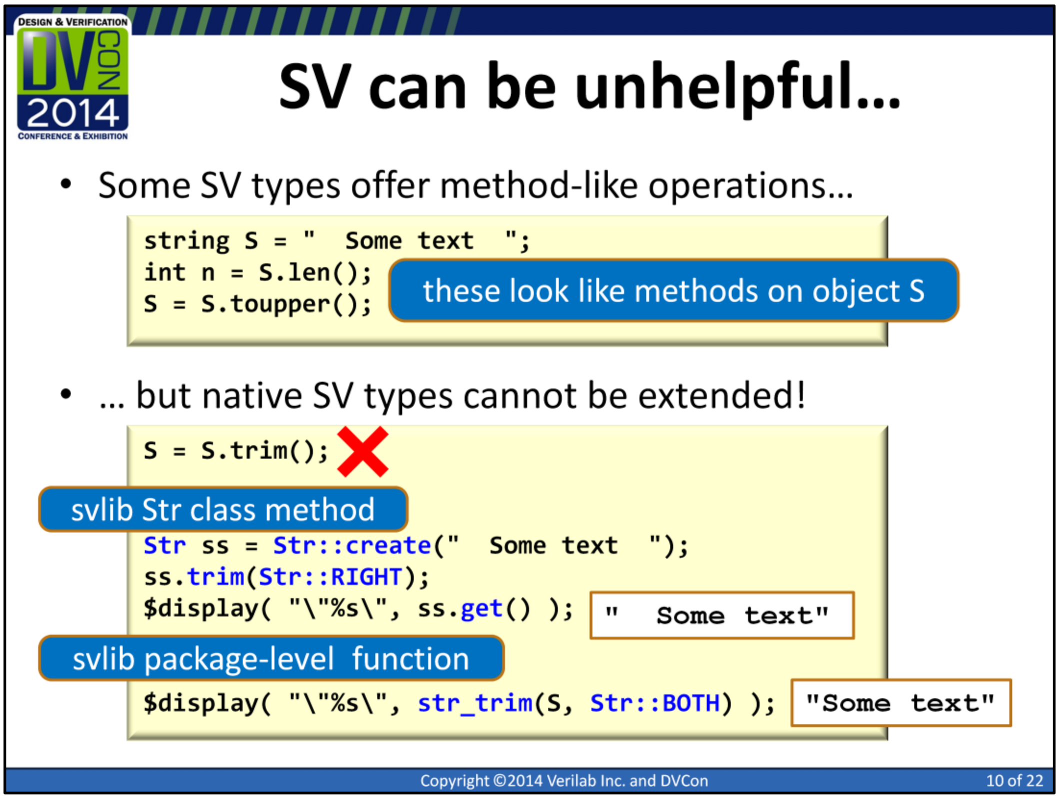
- Choice of initial feature set
- Objects or simple functions?
- Choice of error handling mechanism
- Never disturb random stability

- Guiding principle:

natural and expressive for SV programmers

We have had time here only to give a very brief overview of the features of svlib – you can find much more detail in the distribution.

When designing the library, we took very great care to make it as easy, natural and effective as possible for programmers working in SystemVerilog. The published paper has more to say about our design decisions. Later in this presentation we will examine some of them in more detail.



SV can be unhelpful…

- Some SV types offer method-like operations…

```
string S = "  Some text  ";
int n = S.len();
S = S.toupper();
```

these look like methods on object S

- … but native SV types cannot be extended!

```
S = S.trim();
```

svlib Str class method

```
Str ss = Str::create("  Some text  ");
ss.trim(Str::RIGHT);
$display( "\"%s\"", ss.get() );
```

```
"  Some text"
```

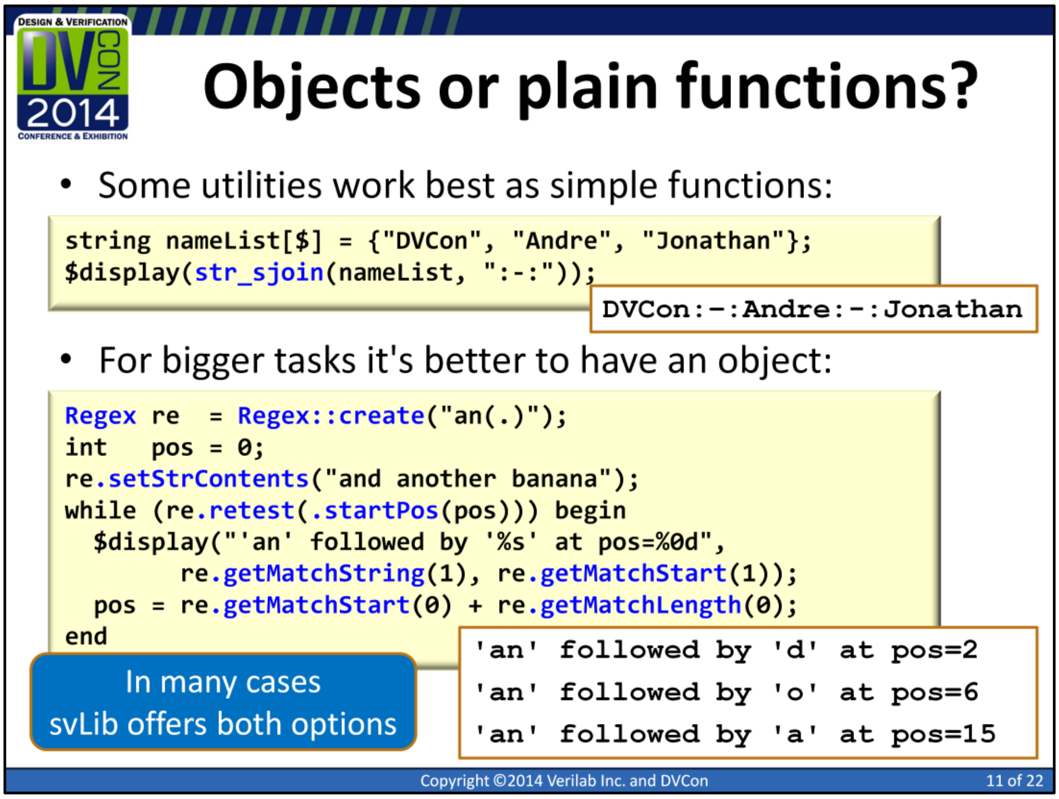
svlib package-level function

```
$display( "\"%s\"", str_trim(S, Str::BOTH) );
```

```
"Some text"
```

The SystemVerilog language itself provided some obstacles to a clean simple design. Here is one interesting example: SystemVerilog has a limited repertoire of string manupulations, expressed as method-like operations on a string variable. We would have very much liked to add more. But we cannot! The string data type is not a class, and cannot be extended. This is very tiresome because it means our library features do not look similar to built-in language features.

We made use of SystemVerilog`s object-oriented programming facilities for some of the library functionality, including the Str (string wrapper) object. For complicated sequences of operations on a string, we think that it is better to construct a Str object (providing a string as its contents) and then work on that object. But this may seem like overkill for simple one-off operations like the string trim function we show here, so svlib also provides some package level functions such as str_trim to make it possible to do string manipulations without first creating a Str object, if you prefer.



Objects or plain functions?

- Some utilities work best as simple functions:

```
string nameList[$] = {"DVCon", "Andre", "Jonathan"};
$display(str_sjoin(nameList, ":-:"));
```

```
DVCon:-:Andre:-:Jonathan
```

- For bigger tasks it's better to have an object:

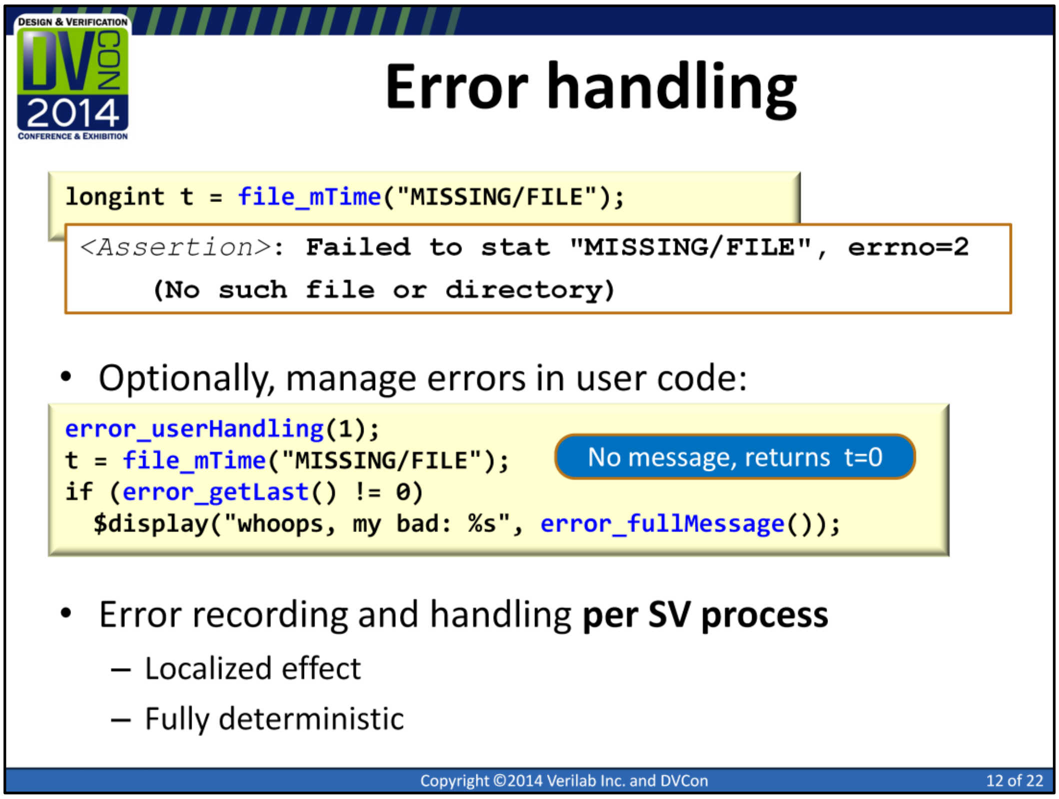
```
Regex re  = Regex::create("an(.)");
int   pos = 0;
re.setStrContents("and another banana");
while (re.retest(.startPos(pos))) begin
  $display("'an' followed by '%s' at pos=%0d",
        re.getMatchString(1), re.getMatchStart(1));
  pos = re.getMatchStart(0) + re.getMatchLength(0);
end
```

```
'an' followed by 'd' at pos=2
'an' followed by 'o' at pos=6
'an' followed by 'a' at pos=15
```

In many cases
svLib offers both options

This slide shows another example of the contrast between package-level functions and the use of SV objects. The string join operation shown in the first box, using function str_join, is probably best done using a package-level function. On the other hand, extracting multiple results from a regular expression match (as in the second example box) is much better expressed as operations on an object, in this case a Regex.

For full details of the methods of class Regex, see the svlib documentation.



Error handling

```
longint t = file_mTime("MISSING/FILE");
```

```
<Assertion>: Failed to stat "MISSING/FILE", errno=2
    (No such file or directory)
```

- Optionally, manage errors in user code:

```
error_userHandling(1);
t = file_mTime("MISSING/FILE");
if (error_getLast() != 0)
  $display("whoops, my bad: %s", error_fullMessage());
```

No message, returns t=0

- Error recording and handling **per SV process**
  - Localized effect
  - Fully deterministic

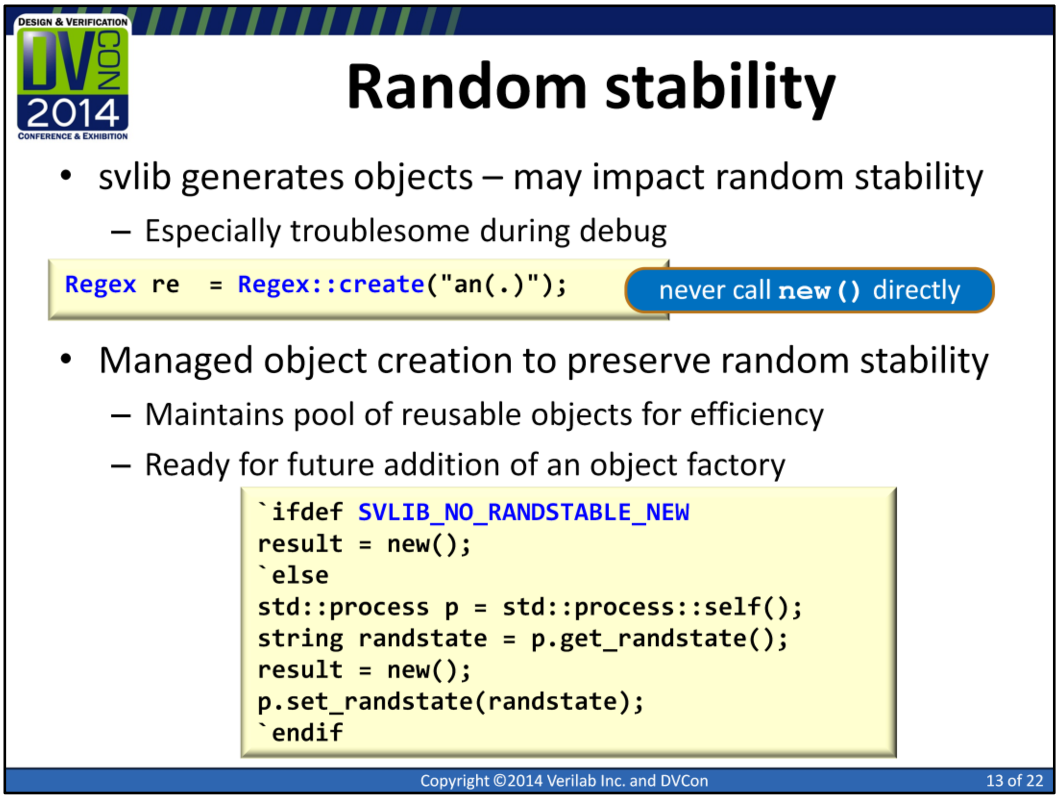
Copyright ©2014 Verilab Inc. and DVCon

12 of 22

Handling of errors was another area that caused us much concern when designing svlib`s API.

Obviously, we did not want errors to go undetected. But if we insist on every function returning a pass/fail result, it makes some parts of the API unnecessarily complicated. Our solution was to provide two different error handling mechanisms. The first, default mechanism is for each error to be reported by an assertion-style error message (the function, meanwhile, returns some appropriate empty result value). Sophisticated users who wish to handle errors for themselves can suppress this mechanism using the error_userHandling function, and then call error_getLast to determine whether the most recent function call caused an error. If an error occurs and the programmer doesn`t handle it, then the next svlib function call will throw a special error to indicate the problem.

This error management mechanism is maintained separately for each SystemVerilog process. In this way the user-handling choice can be localized. At least as important, it also means that errors in one process cannot disturb the error handling and reporting in any other process.



Random stability

- svlib generates objects – may impact random stability
  - Especially troublesome during debug

```
Regex re  = Regex::create("an(.)");
```

never call **new()** directly

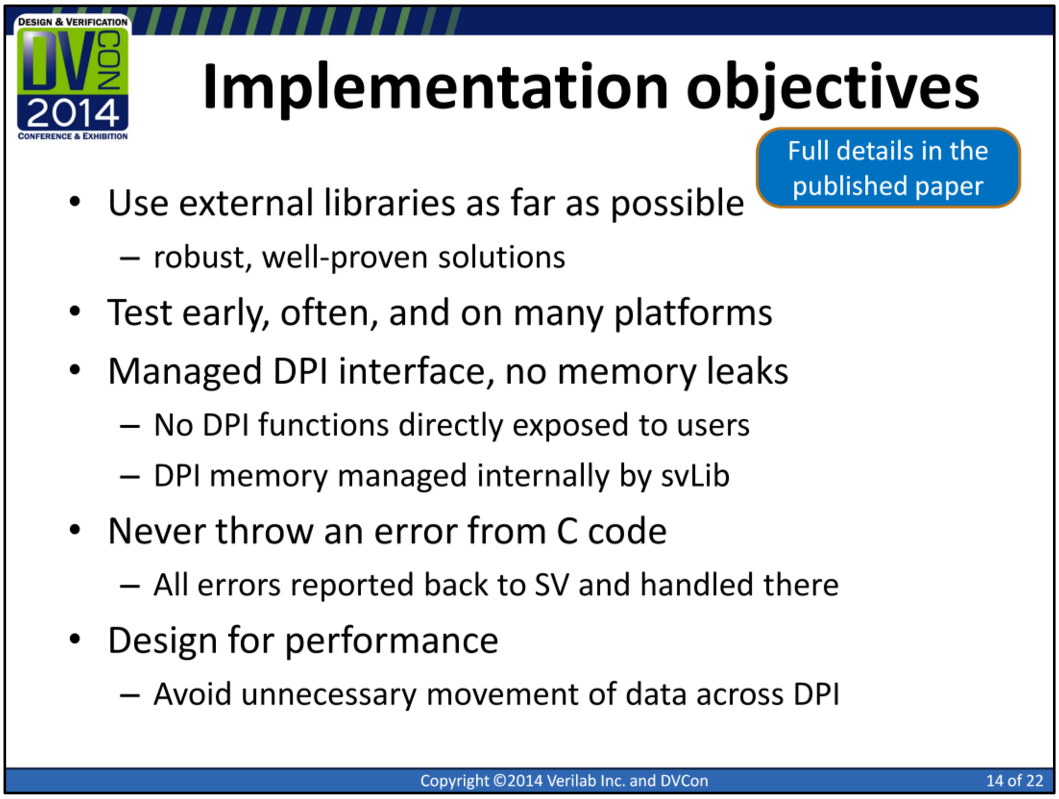
- Managed object creation to preserve random stability
  - Maintains pool of reusable objects for efficiency
  - Ready for future addition of an object factory

```
`ifdef SVLIB_NO_RANDSTABLE_NEW
result = new();
`else
std::process p = std::process::self();
string randstate = p.get_randstate();
result = new();
p.set_randstate(randstate);
`endif
```

13 of 22

Because it is much concerned with string and file handling, we anticipate that svlib will be widely used in debugging code. However, svlib internally creates large numbers of SystemVerilog objects. This could have an adverse performance impact, but – far more importantly – it could disturb the random stability of a SystemVerilog process to which some svlib-based debug code was added, upsetting the randomization of the code you are trying to debug.

We solved this using a two-pronged approach. First, we maintain an internal pool of svlib objects for use within the library itself. Any function that uses an object will get its object from the pool, and return it to the pool when finished. In this way, the number of freshly-created objects is much reduced. Second, all svlib object constructors (new) are protected, so they cannot be called by user code. Instead, users must call a static create method of the appropriate class. This allows us to manage the object pool, and also to create new objects in a controlled manner (shown in the bottom code box here) that does not disturb the calling process`s randomization.

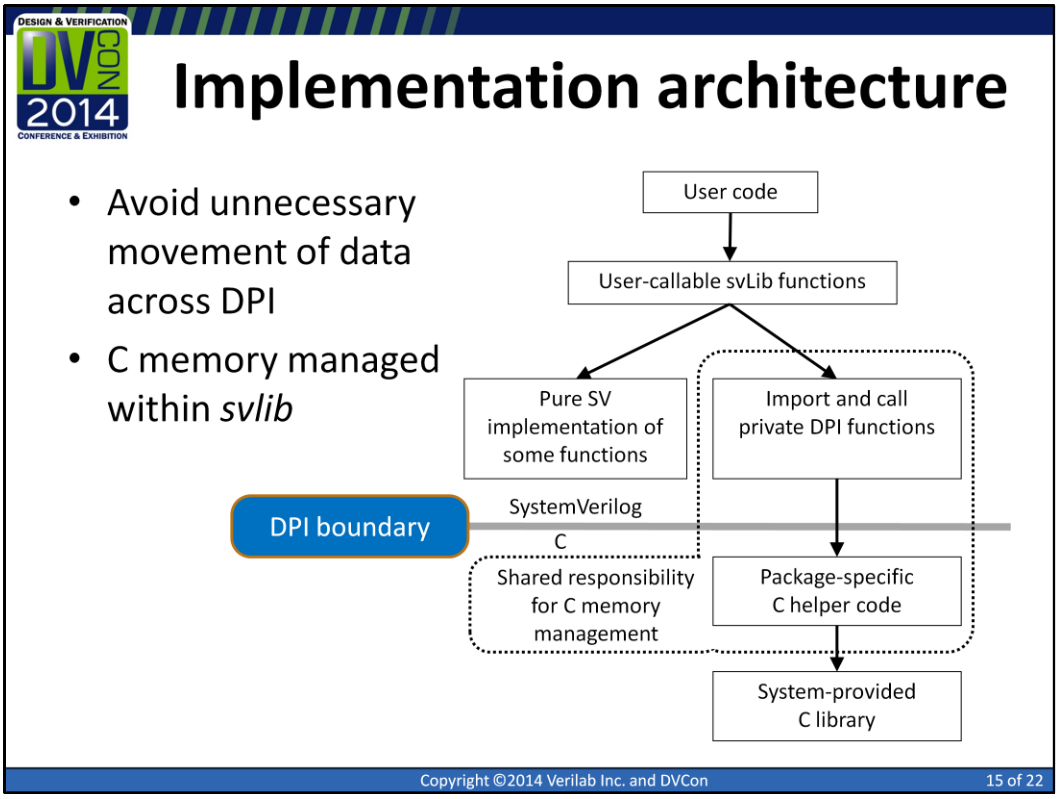


Implementation objectives

Full details in the published paper

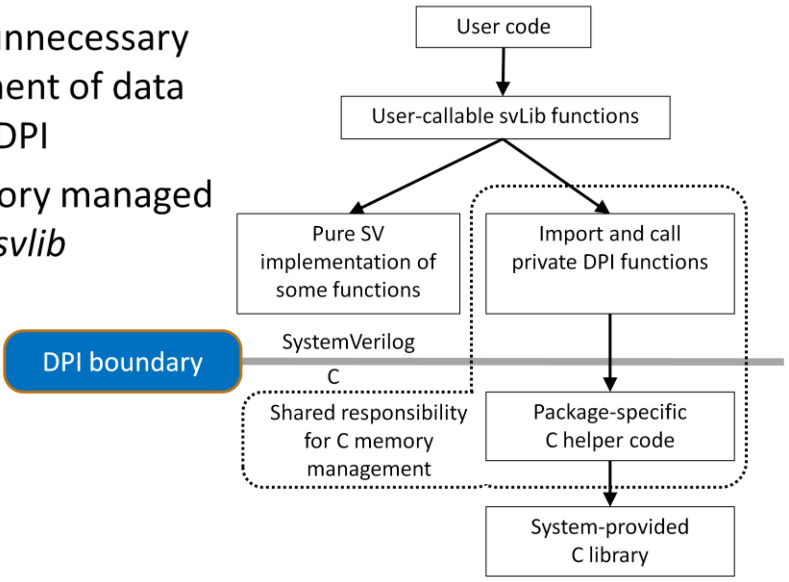
- Use external libraries as far as possible
  - robust, well-proven solutions
- Test early, often, and on many platforms
- Managed DPI interface, no memory leaks
  - No DPI functions directly exposed to users
  - DPI memory managed internally by svLib
- Never throw an error from C code
  - All errors reported back to SV and handled there
- Design for performance
  - Avoid unnecessary movement of data across DPI

As we have outlined in the previous few slides, designing svlib was quite challenging. By contrast, implementing the library was mostly straightforward. However, we tried to keep in mind our concerns about performance, scalability and extensibility. These concerns are described in more detail in the published paper, but this slide highlights the most important areas.

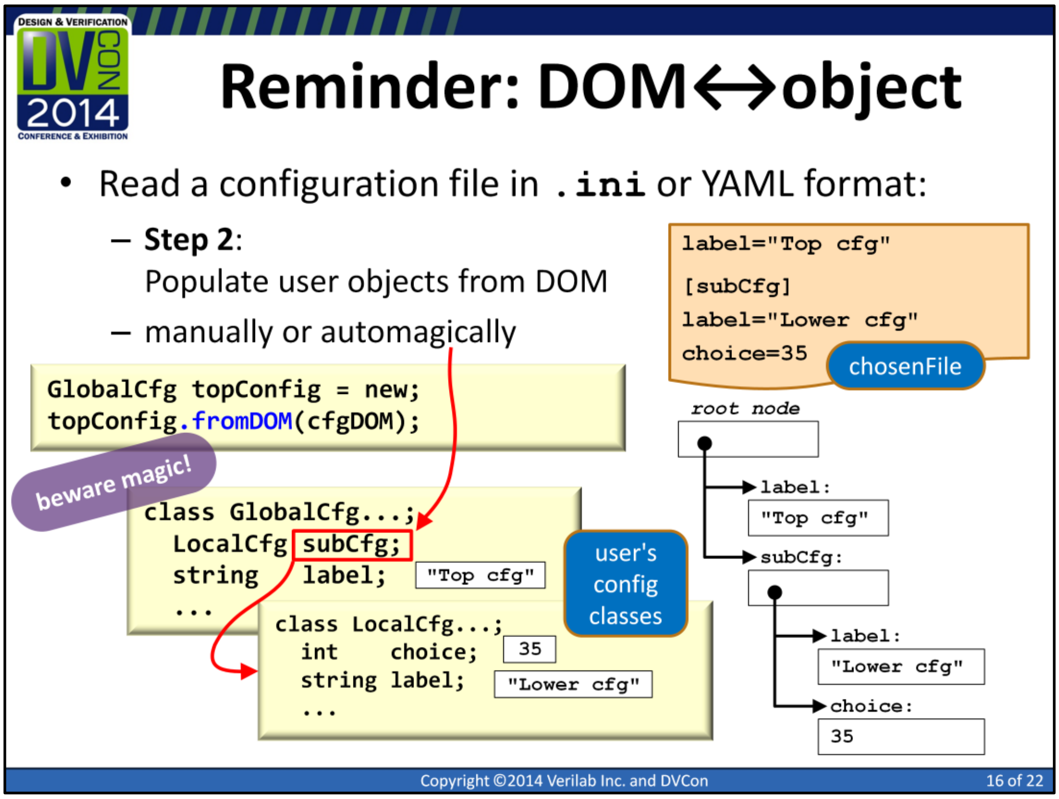


Implementation architecture

- Avoid unnecessary movement of data across DPI
- C memory managed within *svlib*


User code
User-callable svLib functions
Pure SV implementation of some functions
Import and call private DPI functions
DPI boundary
SystemVerilog
C
Shared responsibility for C memory management
Package-specific C helper code
System-provided C library

To allow for future optimizations and improvements, we very much wanted to avoid any user code making calls directly to our C-based DPI functions. Consequently, the overall architecture of svlib is as shown in the diagram above. All C functions are hidden behind a management layer in SystemVerilog. Package and naming conventions are used to enforce this layering.



Reminder: DOM↔object

- Read a configuration file in `.ini` or YAML format:
  - **Step 2**: Populate user objects from DOM
  - manually or automagically

```
label="Top cfg"
[subCfg]
label="Lower cfg"
choice=35
```

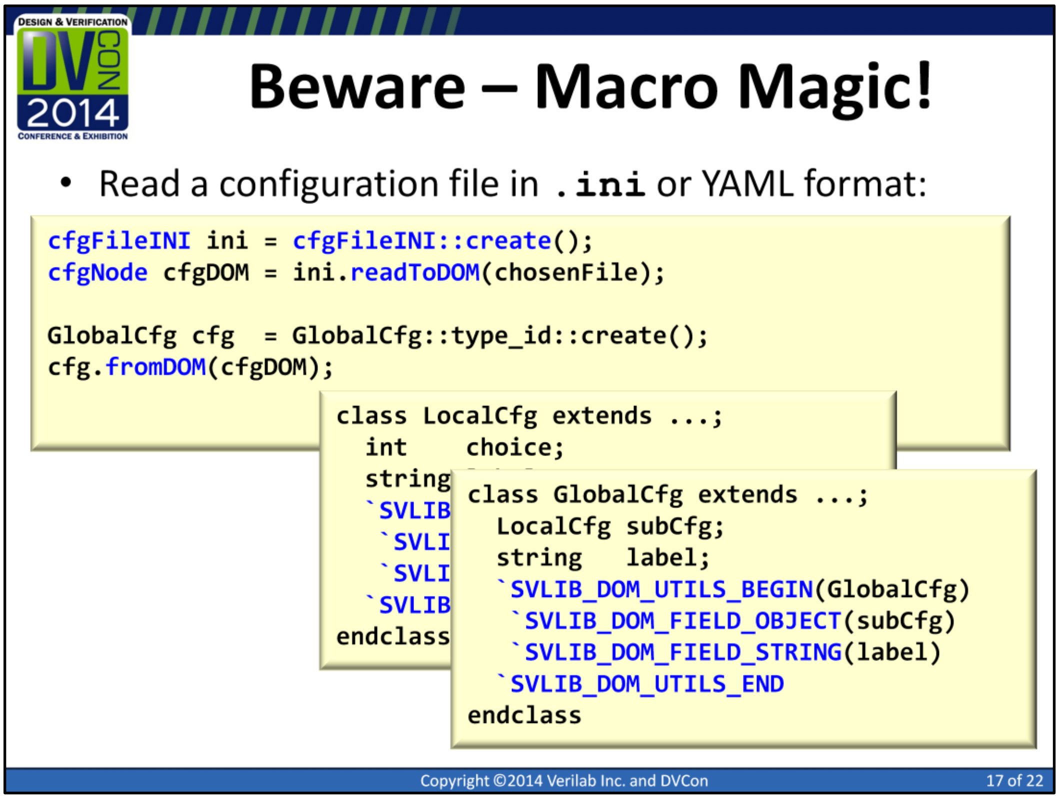
```
GlobalCfg topConfig = new;
topConfig.fromDOM(cfgDOM);
```

```
class GlobalCfg...;
  LocalCfg subCfg;
  string   label;   "Top cfg"
  ...
```

```
class LocalCfg...;
  int    choice;    35
  string label;     "Lower cfg"
  ...
```

Now it`s time to revisit the magic we mentioned earlier, allowing us to populate an arbitrary user-designed object from the contents of a DOM (document object model) class.

This slide is a reminder of the example we presented earlier. In the next few slides we`ll show how it is done.



Beware – Macro Magic!

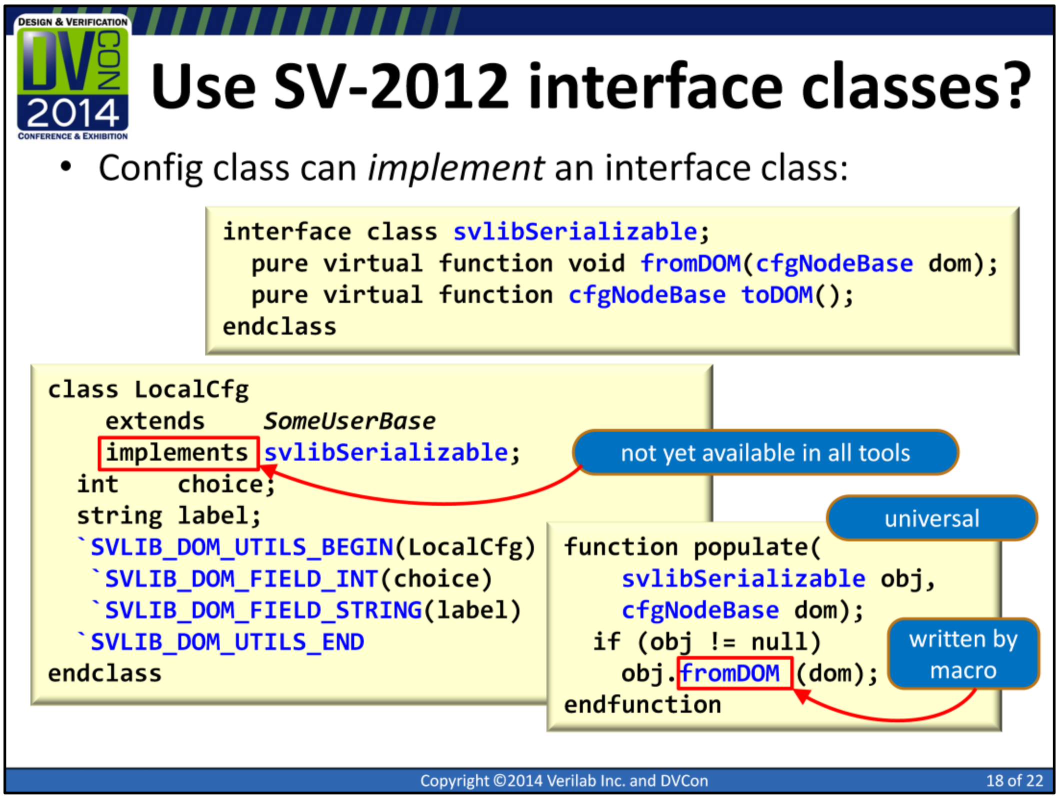
- Read a configuration file in `.ini` or YAML format:

```
cfgFileINI ini = cfgFileINI::create();
cfgNode cfgDOM = ini.readToDOM(chosenFile);

GlobalCfg cfg  = GlobalCfg::type_id::create();
cfg.fromDOM(cfgDOM);
```

```
class LocalCfg extends ...;
  int    choice;
  string
  `SVLIB
   `SVLI
   `SVLI
  `SVLIB
endclass
```

```
class GlobalCfg extends ...;
  LocalCfg subCfg;
  string   label;
  `SVLIB_DOM_UTILS_BEGIN(GlobalCfg)
   `SVLIB_DOM_FIELD_OBJECT(subCfg)
   `SVLIB_DOM_FIELD_STRING(label)
  `SVLIB_DOM_UTILS_END
endclass
```

Taking our cue from the UVM`s field automation macros. we provide macros that a user can add to their own classes to automate copying of their class`s data members to or from a suitable svlib DOM. Only the data members that you want to copy to/from a DOM should be listed in the macros. These macros, working together, implement two new methods of your class: fromDOM and toDOM.



Use SV-2012 interface classes?

- Config class can *implement* an interface class:

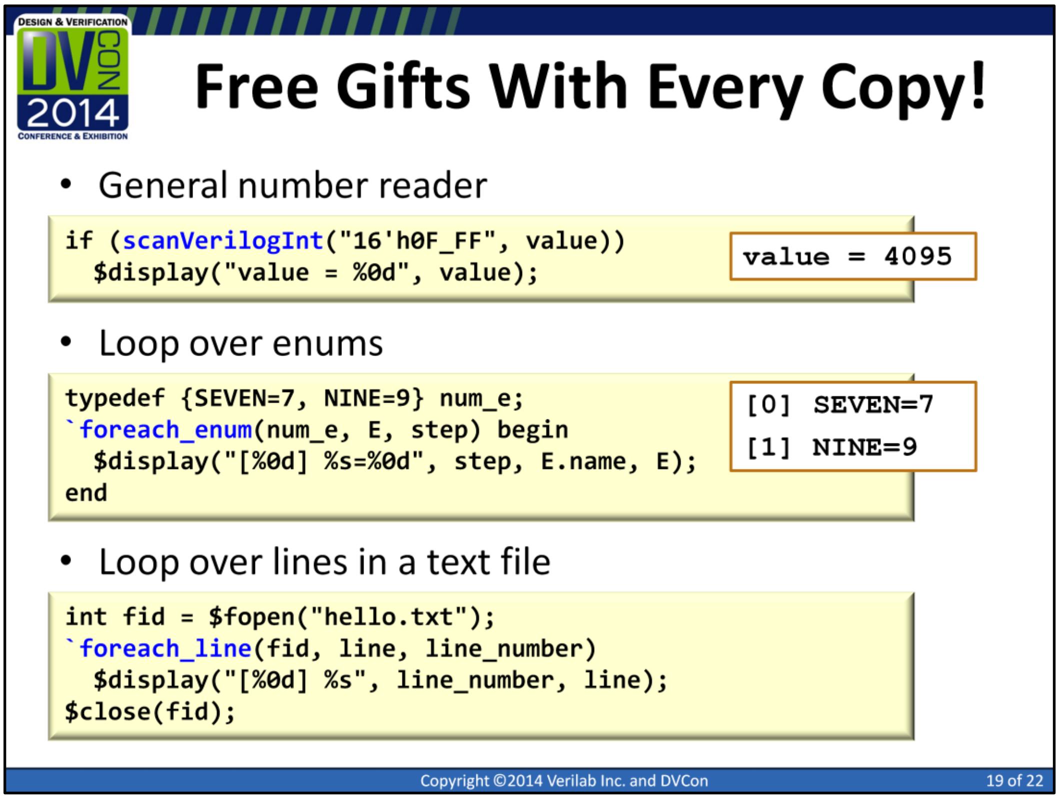
```
interface class svlibSerializable;
  pure virtual function void fromDOM(cfgNodeBase dom);
  pure virtual function cfgNodeBase toDOM();
endclass
```

```
class LocalCfg
    extends    SomeUserBase
    implements svlibSerializable;
  int    choice;
  string label;
  `SVLIB_DOM_UTILS_BEGIN(LocalCfg)
   `SVLIB_DOM_FIELD_INT(choice)
   `SVLIB_DOM_FIELD_STRING(label)
  `SVLIB_DOM_UTILS_END
endclass
```

```
function populate(
    svlibSerializable obj,
    cfgNodeBase dom);
  if (obj != null)
    obj.fromDOM (dom);
endfunction
```

The macro machinery described in the previous slide is convenient, but it has a major limitation. We cannot expect svlib users to derive all their data classes from some svlib base class – that would be excessively selfish, and would clash with UVM or other methodology requirements. Consequently, each class will have its own entirely independent fromDOM and toDOM methods. But users probably want to be able to write universal machinery that can perform these operations on any of their DOM-equipped classes. The latest 2012 revision of SystemVerilog adds an interesting new feature, interface classes, that support precisely this requirement. The example class LocalCFG shown here is derived from some other base class (perhaps uvm_object?) but it also implements the interface class svlibSerializable. This interface class specifies only some pure virtual methods; it has no data members or other methods. But now we can write function populate() that can operate on any object that implements this interface class.

Unfortunately, support for this feature is not yet available in all simulators – so, reluctantly, we have not added it to svlib. As soon as simulator support is universally available, we will retrofit it.



Free Gifts With Every Copy!

- General number reader

```
if (scanVerilogInt("16'h0F_FF", value))
  $display("value = %0d", value);
```

```
value = 4095
```

- Loop over enums

```
typedef {SEVEN=7, NINE=9} num_e;
`foreach_enum(num_e, E, step) begin
  $display("[%0d] %s=%0d", step, E.name, E);
end
```

```
[0] SEVEN=7
[1] NINE=9
```

- Loop over lines in a text file

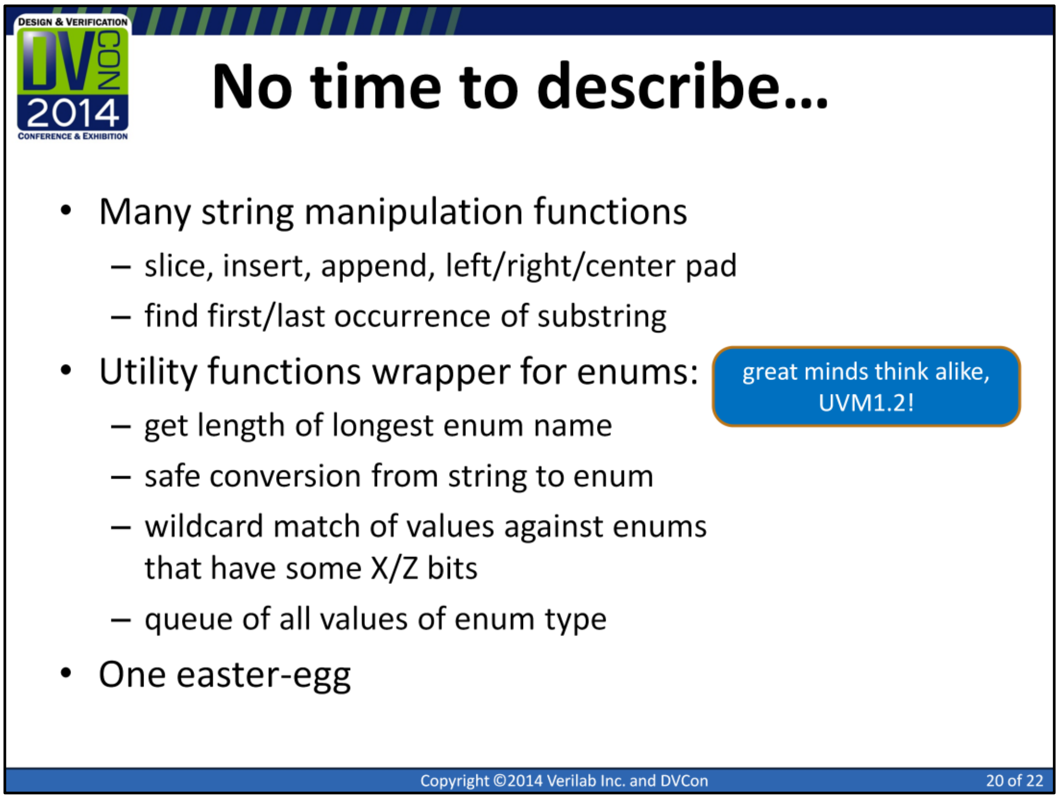
```
int fid = $fopen("hello.txt");
`foreach_line(fid, line, line_number)
  $display("[%0d] %s", line_number, line);
$close(fid);
```

svlib has a number of other useful features that we added because we like them.

scanVerilogInt plugs a minor gap in the existing $scanf family of functions, allowing you to read integral values in the full Verilog based format.

foreach_enum is a convenience macro to loop over all the values of an enumeration type, in the same way as a foreach loop.

foreach_line similarly loops over each line of text in a file.

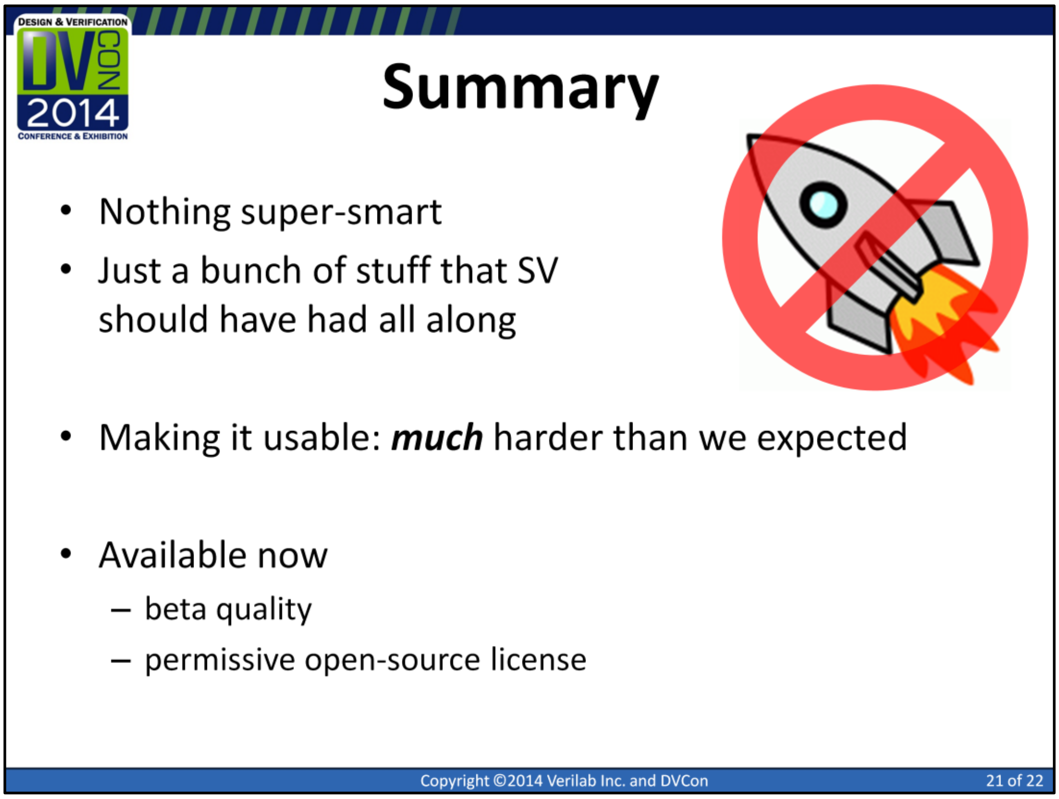


No time to describe…

- Many string manipulation functions
  - slice, insert, append, left/right/center pad
  - find first/last occurrence of substring
- Utility functions wrapper for enums:
  - get length of longest enum name
  - safe conversion from string to enum
  - wildcard match of values against enums that have some X/Z bits
  - queue of all values of enum type
- One easter-egg

great minds think alike, UVM1.2!

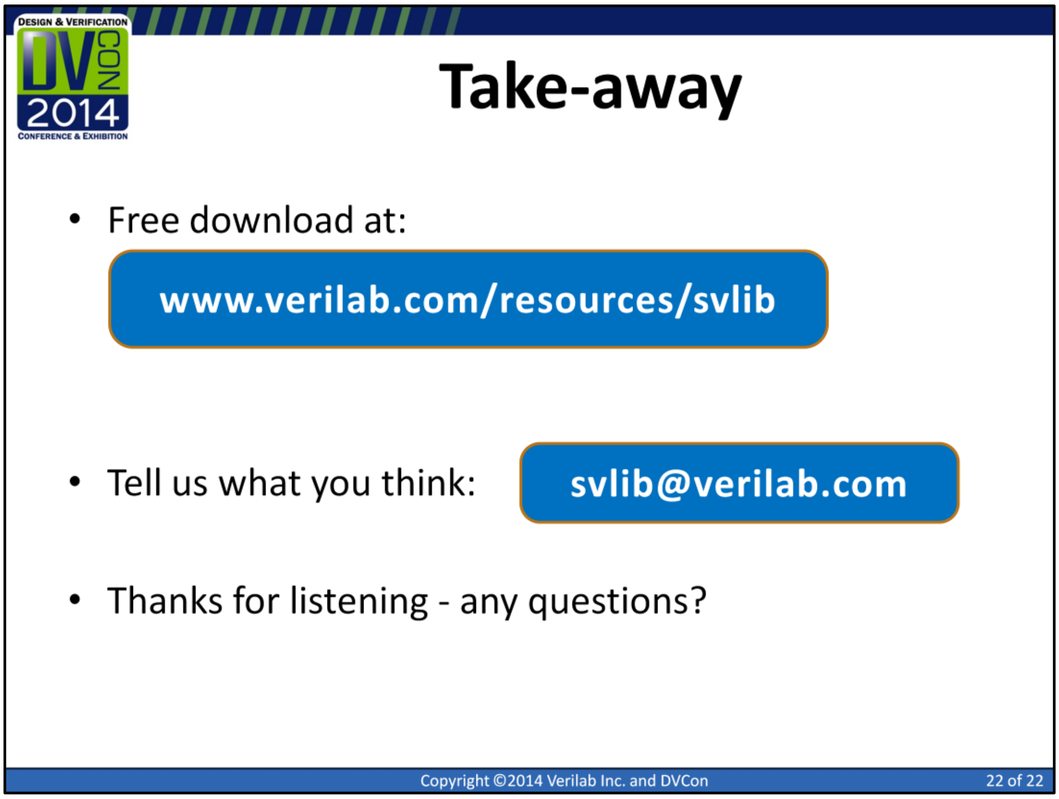
svlib has many other features that don`t fit into this presentation!



Summary

- Nothing super-smart
- Just a bunch of stuff that SV should have had all along
- Making it usable: ***much*** harder than we expected
- Available now
  - beta quality
  - permissive open-source license

In summary: svlib is just a collection of features that we wish were in SystemVerilog from the outset.



Take-away

- Free download at:

  www.verilab.com/resources/svlib

- Tell us what you think: **svlib@verilab.com**
- Thanks for listening - any questions?

You are welcome to download the package, documentation and paper from our website. We have set up a special email alias and we welcome any feedback on svlib – please tell us what you liked and what you did not like, so we can work to make it better.