

Formal Verification for SystemC/C++ Designs

Vlada Kalinic, OneSpin: A Siemens Business

SIEMENS



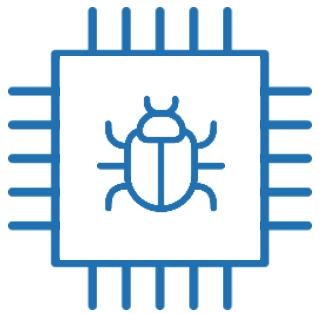
Agenda

- Introductions
- Overview of HLS usage, current challenges, opportunities
- OneSpin – SystemC DV Inspect and Verify Overview
- Q&A

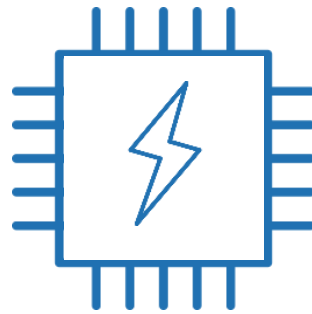
IC Integrity

Functionally Correct, Safe, Secure, and Trusted SoCs/ASICs/FPGAs

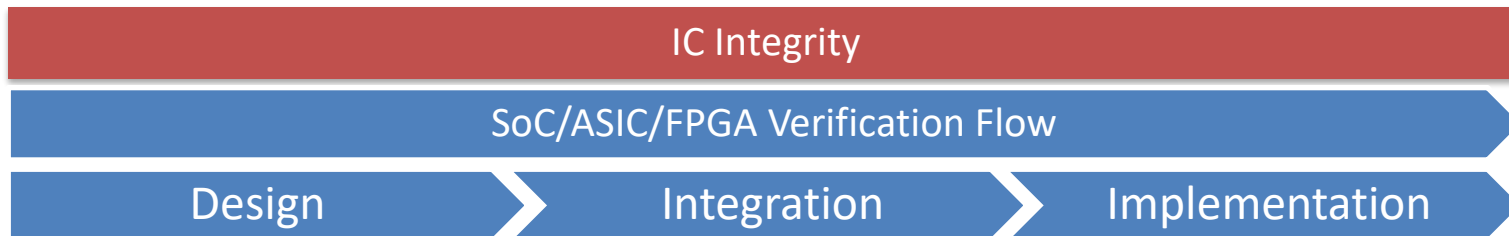
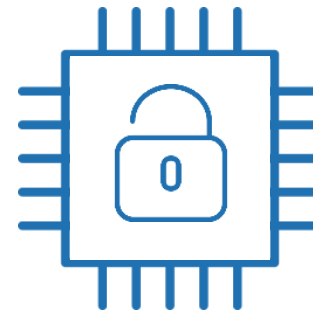
Functional
Correctness



Safety



Trust and
Security



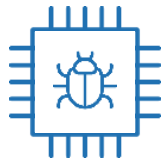
OneSpin: A Siemens Business provides certified **IC Integrity Verification Solutions** to develop functionally correct, safe, secure, and trusted integrated circuits.

Leading-Edge Formal Technology

Targeting Critical Hardware Verification Challenges

Functional Correctness

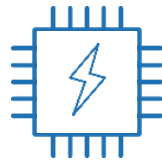
Rigorous coverage-driven functional verification from block to chip, leveraging formal technology



- Design Exploration
- Protocol Violations
- Integrate Formal/Sim Coverage
- End-to-End User Assertions
- HLS/SystemC Verification
- Synthesis/P&R Errors

Safety

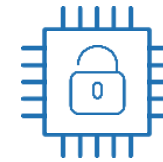
Safety analysis and higher diagnostic coverage to meet strict certification requirements



- FMEDA of Complex SoCs
- Failure Mode Distribution
- Avoid Excessive Fault Simulations
- Measure Diagnostic Coverage
- ISO 26262 Compliance
- Tool Qualification

Trust and Security

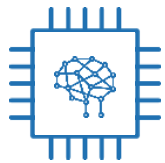
Automated detection of RTL Trojans and hardware vulnerabilities to adversary attacks



- Denial of Service
- Data Leakage
- Privileges Escalation
- Data Integrity/Confidentiality
- Hardware Backdoors
- Hardware Trojans

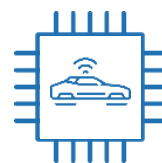
OneSpin 360[®] Formal Platform

Heterogeneous Computing



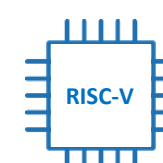
Thorough verification of complex SoC platforms used for 5G wireless, IoT, and AI applications

Automotive and Industrial



Systematic bug elimination and metrics on proper handling of random errors in the field

RISC-V



Efficient and complete verification, including custom extensions. Compliance to ISA.

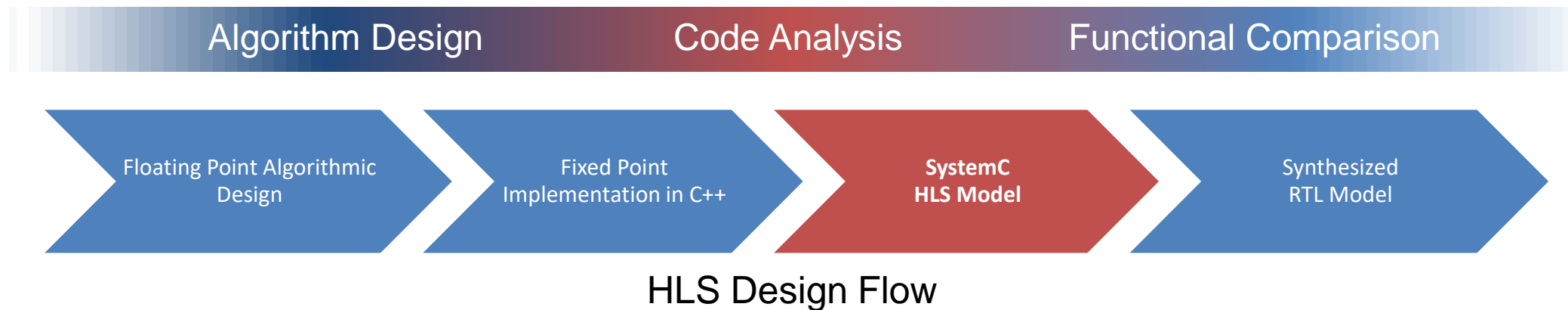
OneSpin Solutions and Services

SystemC with HLS Typical Issues

Imposing Hardware Constraints on C++

Using SystemC for HLS Modeling creates new problems and opportunities

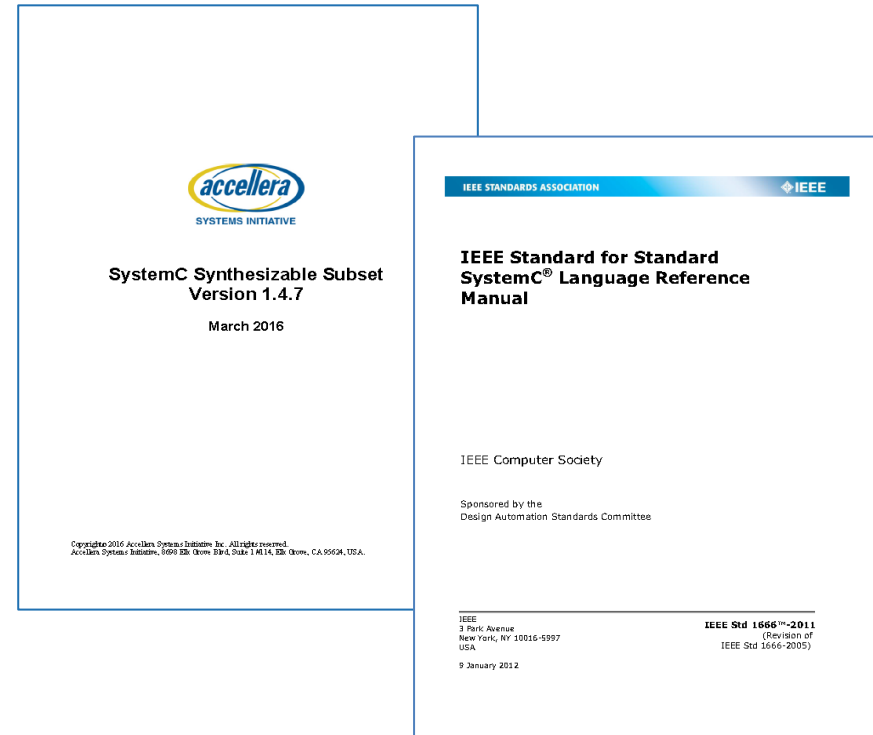
- Algorithm implementation issues in SystemC
- SystemC language related code problems and ambiguity in the code
 - There are undefined operations in the SystemC Standard
- Functional Consistency Checking of SystemC vs. RTL



The Standards Do Not Help

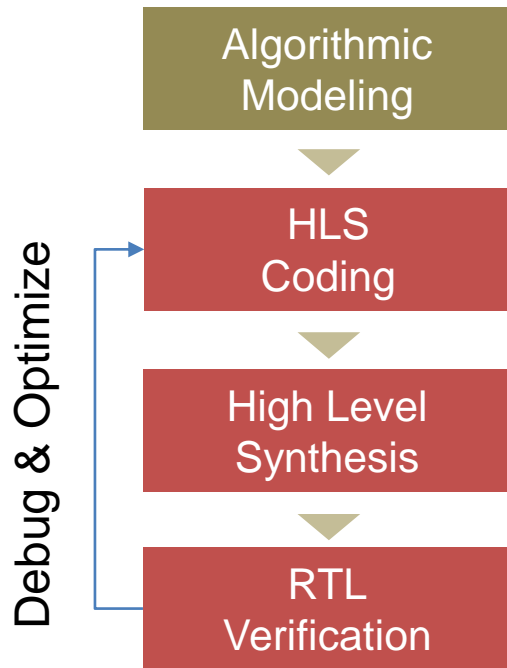
C++ Not Built for Hardware Descriptions

- IEEE 1666 SystemC Standard
 - 25+ occurrences of “unspecified”
 - 50+ occurrences of “undefined”
 - 150+ occurrences of “implementation defined”
- Accellera Synthesizable Subset
 - ~20 occurrences of “undefined”, “unspecified”, “implementation defined”
 - OneSpin supports C++ 14 version



OneSpin: Advanced Verification for HLS

Current HLS Flow



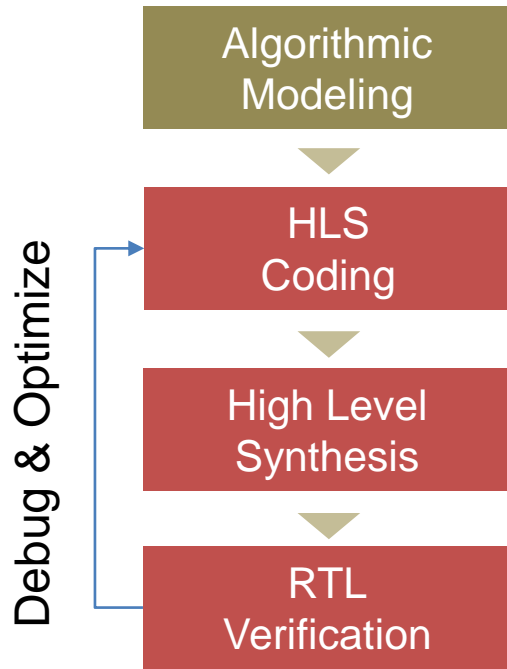
- Challenges

- Limited useful feedback from HLS for coding style
- Certain coding mistakes can cause simulation mismatches that are extremely difficult to debug
- Optimization loop is long and somewhat ad-hoc
- Garbage in, garbage out

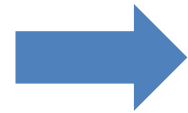
OneSpin: Advanced Verification for HLS

- Opportunities to Improve Design Flow

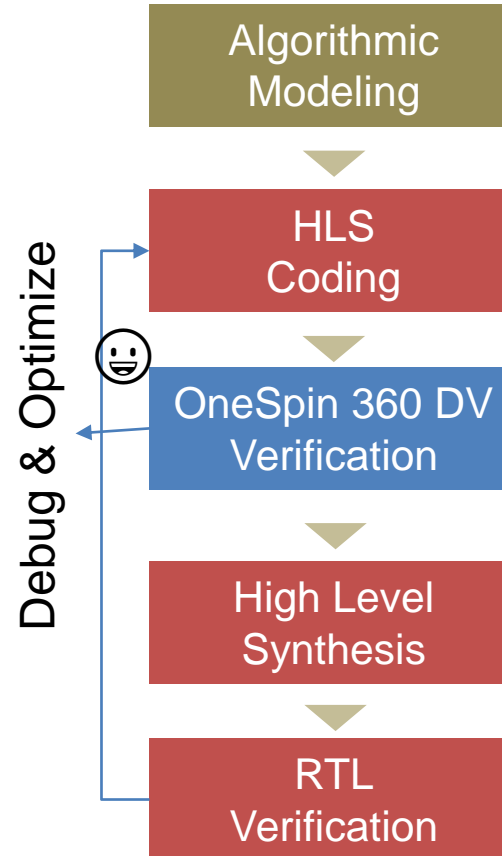
Current HLS Flow



Adding OneSpin



Improved HLS Flow



- Early verification and bug detection
- Better SystemC verification
 - Automated and exhaustive
 - Formal checking – not simple linting
 - Clearer messages & direction to improve code
 - Comprehensive coverage metrics
- Faster runtime and iteration loop
- Check over SystemC common issues on the original SystemC code (as undefined operation in Standard)

Formal Autochecks
Automated Apps
SVA / Assertions

Deploying the OneSpin Products

DV-Inspect & DV-Verify for SystemC & RTL

Automatic Formal Analysis

DV-Inspect

- Structural Analysis
- Linting
- Initialization & Reset
- Overflow and Array OOB
- Activation & Reachability
- Arithmetic Precision
- Race Conditions

Tool Guided Verification

DV-Verify Apps

- Design Exploration
- UMR & X-Propagation
- Protocol Verification IP
- Scoreboard

UMR = Uninitialized Memory Read

Assertion Based Verification

DV-Verify Formal ABV

- SV-Assertions, C-Assert
- Cover Points
- Observation Coverage

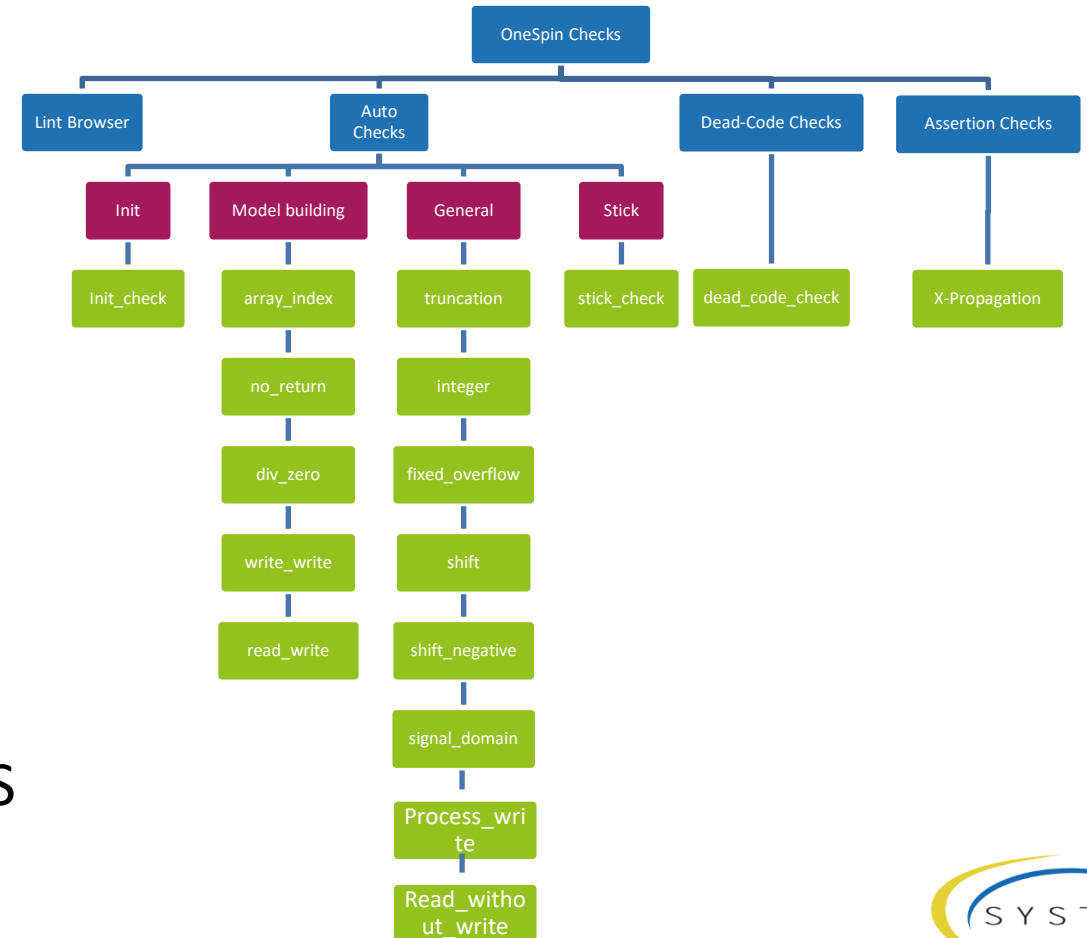
Easy Adoption & Increasing Value

OneSpin 360 DV High-Level Verification

Design Verification Solution for C++/SystemC HLS Code

- Values (Pros over RTL verification)
 - Eliminate design bugs before HLS synthesis
 - Start verification much earlier in the process
 - Reduce simulation effort in SystemC and RTL
 - Optimize HLS input code before synthesis
 - Both C++ or SystemC languages for HLS

OneSpin 360 DV-Inspect for SystemC



OneSpin Formal Inspection

SystemC Code Apps & Checks



- Automatic identification of SystemC problems and coding style issues
- No need for testbench writing
- **Formal checks** – not just linting
- Problems easily debugged prior to synthesis

Structure (Easy Lint)	Safety Checks (Assertion Synthesis)			Activation (Coverage)
Mismatch/port /wire	Runtime Errors	Sim-Synth Issue	Safe Function	Dead code checks
Signal trunc / no sink	Array index	Initialization	Arithmetic overflow	Stuck signal (toggle test)
Sensitivity list issues	Function without return	X-Propagation	Redundant bits	FSM trans and states
Unused signal / param	Division by 0	Write-write race	Arithmetic shifts	MORE...



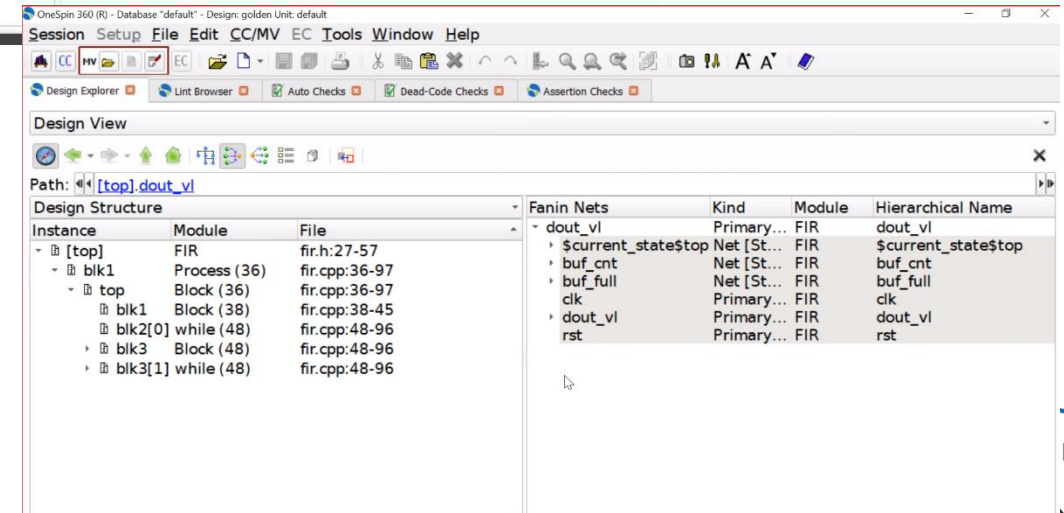
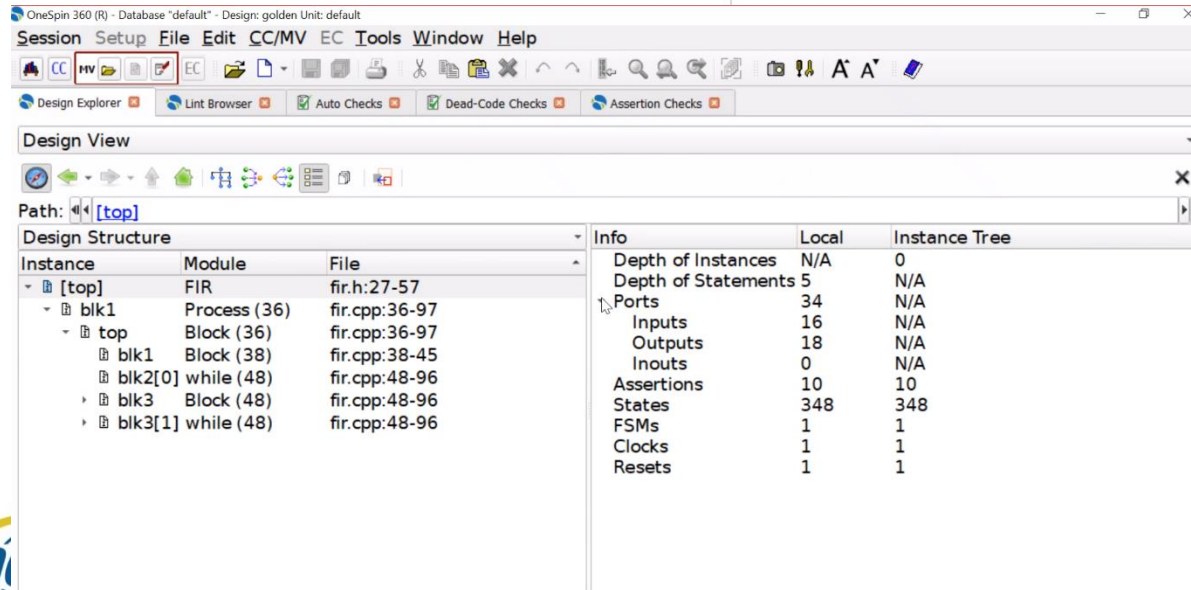
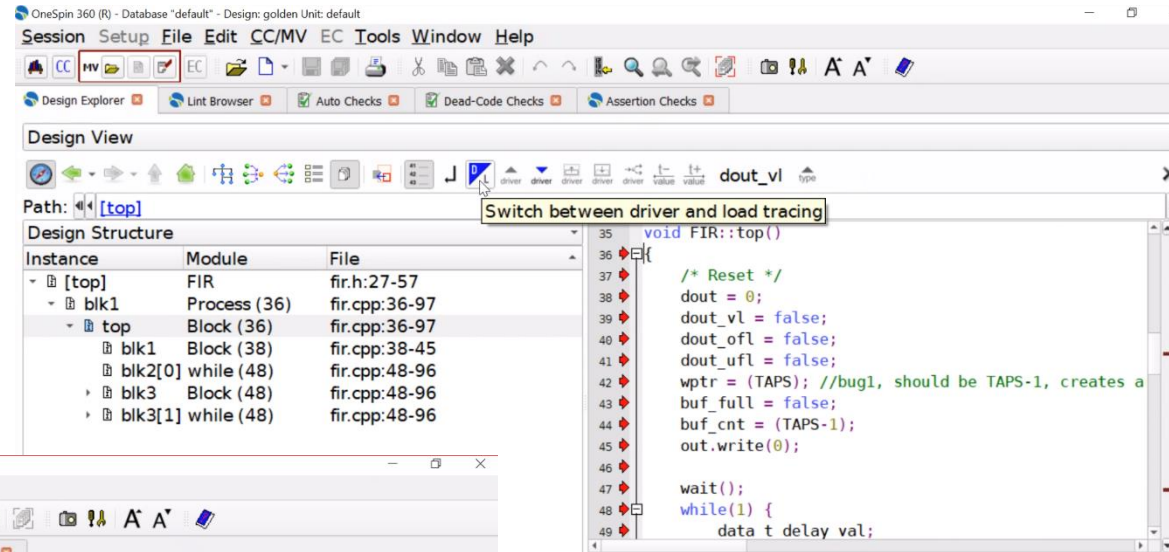
SystemC Inspect Automated Checks

Tab	Autocheck	Explanation
Init	<u>init</u>	Initialization checks are created for each non-redundant state signal and primary output of the current unit. An initialization check tests whether the corresponding signal is set to a uniquely determined value when applying the reset sequence of the unit.
ModelBuilding	<u>array_index</u>	An array index violation occurs if an array is accessed using an index which exceeds the array bounds. Array index checks check for static and dynamic violations in all array accesses occurring in the HDL source code.
ModelBuilding	<u>div_zero</u>	Division-By-Zero checks are generated for all arithmetic divisions occurring in Verilog, SystemC and VHDL source code, checking whether or not the divisor is always different from zero. These checks are also generated in Verilog and SystemC for modulo operations with a zero base and for pow operations on zero with a negative exponent.
ModelBuilding	<u>no_return</u>	Function-Without-Return checks test whether each possible control path through a function ends with a return statement.
General	<u>shift_negative</u>	Checks whether a shift with a negative direction occurs. Cannot occur in SystemVerilog, since there shift counts are always treated as unsigned integers.
ModelBuilding	<u>signal_domain</u>	Signal domain checks investigate whether state bits of the unit can take a value other than zero or one, e.g. 'X' or 'Z'.
ModelBuilding	<u>write_write</u>	In Verilog and SystemC designs, it is possible that write-write races occur among different processes. In VHDL, a write-write race check is generated if a racing condition for a shared variable may occur.
ModelBuilding	<u>read_write</u>	In Verilog and SystemC designs, it is possible that read-write races occur among different processes if blocking assignments are used. In VHDL, a read-write race check is generated if a racing condition for a shared variable may occur.

Tab	Autocheck	Explanation
General	<u>fixed_overflow</u>	Checks for overflows in fixed_float implementations in VHDL and SystemC.
General	<u>Integer</u>	Integer checks are created for each signed or unsigned integer signal of the current unit. An integer check tests whether there are redundant bits in the signal.
General	<u>shift</u>	A signal can be accidentally set to zero by logically shifting its value too many times in the same direction. For each shift operation occurring in the source code, a shift check is created, checking whether or not such unintended behavior may occur.
General	<u>process_write</u>	In SystemC designs, it is possible that write-write race occur within same process. For all possibly affected signals, a process-write check is generated, investigating whether such incident can happen.
General	<u>read_without_write</u>	In SystemC designs, it is possible that a read of a signal is performed before the signal being written for the first time. For all possibly affected signals, a read_without_write check is generated, investigating whether such incident can happen.
General	<u>truncation</u>	If the result of an integral operation is used in a context, that does not match the self-determined size or signedness of the operation, then relevant bits may be lost.
Dead Code	<u>dead_code</u>	A line of code is called dead code if it is not visited in any execution trace. Lines can be unreachable, for example, if the condition of an enclosing control structure never becomes true, thus always preventing it from being executed.
Stick	<u>stick</u>	Stick checks test the unit for constant bits in signals.
Assertion Checks	<u>x_checking_setup</u> <u>x_checking</u>	X-Propagation Analysis app provides a robust and effective circuit analysis that highlights all the issues in a design that could lead to X state propagations without reliance on simulation test stimulus.

Design Exploration

- Design browser
- Full debugger



Handling SystemC Initialization

Unpredictable Reset States

Automatic variable initialization in SystemC
(due to C++ mother language)

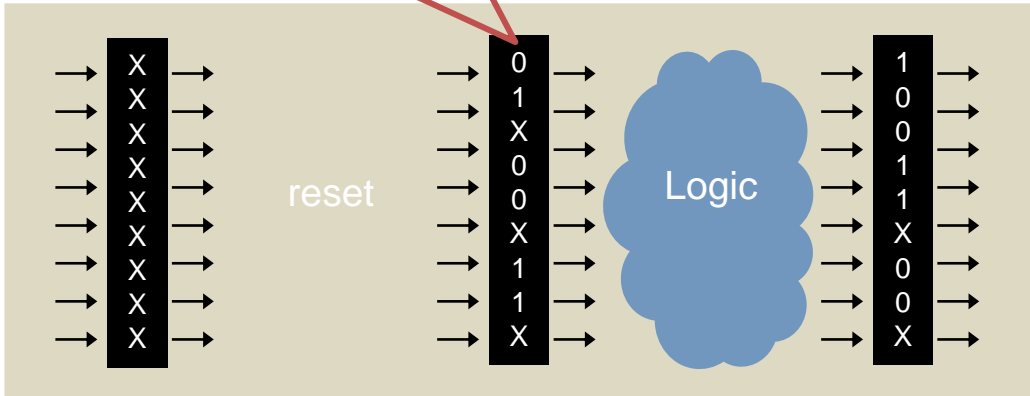
- All “sc_” datatypes automatically initialized to default value

However, synthesizable subset standard states:

- Module constructor initializations ignored
- Reason: Reset behavior under user’s control

Inevitable Sim/Synth mismatches hard to debug using simulation

- ✓ Checks which registers are initialized
- ✓ Check (intentionally) undefined reg effect
- ✓ Switch between sim & synth semantics



Init checks

Autochecks categories

- **A) Decription:** Initialization checks are created for each non-redundant state signal and primary output of the current unit. An initialization check tests whether the corresponding signal is set to a uniquely determined value when applying the reset sequence of the unit.
- **B) Command to execute only these checks:**

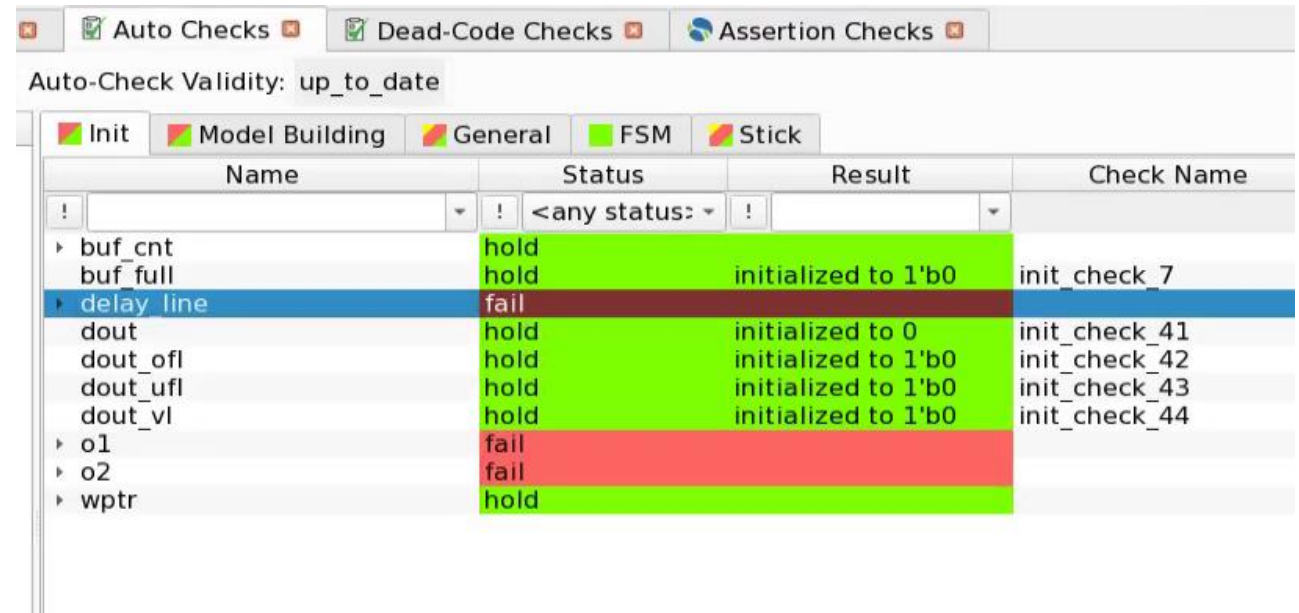
`check_consistency -category init`

- **C) Example of the code:**

```
data_t delay_line[TAPS];  
...  
/* Reset */  
dout = 0;  
dout_vl = false;  
dout_ofl = false;  
dout_ufl = false;  
wptr = (TAPS);  
buf_full = false;  
buf_cnt = (TAPS-1);
```

Signal delay_line is not reset.

- **D) Debug:** Debugging is done by clicking "Go to Source" button
- **E) Reason to fix:** In case this issue persists, uninitialized values might cause 'X' values and to occur at some of the design's outputs.

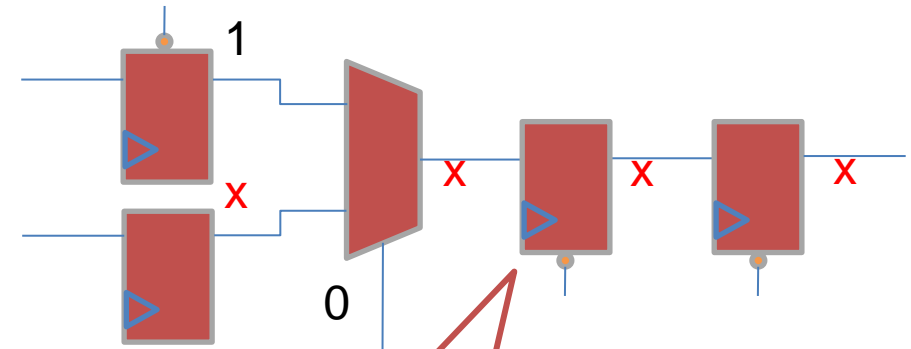


Name	Status	Result	Check Name
! <any status>	!	!	
buf_cnt	hold		
buf_full	hold	initialized to 1'b0	init_check_7
delay_line	fail		
dout	hold	initialized to 0	init_check_41
dout_ofl	hold	initialized to 1'b0	init_check_42
dout_ufl	hold	initialized to 1'b0	init_check_43
dout_vl	hold	initialized to 1'b0	init_check_44
o1	fail		
o2	fail		
wptr	hold		

Undefined Value Propagation

No X-State in SystemC

- Are all registers initialized?
 - Uninitialized registers sources of X instability
- Other sources of X
 - Undefined operations
 - Multiple drivers
- If Xs occur, will this have a bad effect?
- Solutions?
 - SystemC Simulator has no notion of undefined values or RTL semantics
 - Formal can exhaustively analyze all conditions under which an X can propagate



OneSpin 360 DV SystemC

- ✓ Handles all sources of Xs
- ✓ Automated App to track X propagation
- ✓ Manual assertions (if Xs are allowed temporarily)

Undefined Operations

Example Array Out-Of-Bounds Access

Simulation

- Array address maybe larger than number of elements but no range checking
- Undefined behavior with diverse effects
- C++ checking tools slow and cumbersome
 - Std::vector not possible
- Trivial bugs are hard to find and debug

```
sc_uint<8> mem[8][16];
      :
if(x>=16) { x = 15 };
if(y>=8) { y = 7 };
mem[x][y] = ...;
```

- Simulation does not complain and runs fine!
- DV-Inspect reports range violation error.

OneSpin 360 DV-Inspect

- ✓ Exhaustive analysis
- ✓ Precise error location
- ✓ Easy debug

```
sc_signal<sc_uint<10> > intArr[10];
      :
int b = (large ? 10 : 5);
for(int i = 0; i <= b; ++i)
    intArr[i].write(0);
```

- Simulation does not complain but may crash if b = 10!
- DV-Inspect reports range violation error with b = 10 if 'large' is possible.

Array index violation

Autochecks categories

- A) Description:**

An array index violation occurs if an array is accessed using an index which exceeds the array bounds. Array index checks check for static and dynamic violations in all array accesses occurring in the HDL source code.

- B) Command to execute only these checks:**

`check_consistency -category array_index`

- C) Example of the code:**

```
const int TAPS=27;
data_t delay_line[TAPS];
delay_line[wptr] = din_reg;
```

Array delay_line' which contain 27 elements (0 to 26), and it the code it accesses on 27th element.

- D) Debug:**

Debugging is possible by clicking "Debug" button. The debugger points to line where the problem occurs. Example is in the next slide.

- E) Reason to fix:**

Another one of many code issues in SystemC is the array out of bounds problem. In case this issue persists, "delay_line" could have some undesired values. Index-out-of-bound can generate 'X' values or some unknown and incorrect design behavior. In hardware, a memory or register array may be addressed by another register or counter value leading to an accidental value. Again this can be hard to track in SystemC using a simulator. But formal is ideal of quickly identifying these problems, however the code structure appears that creates the issue.

Name	Category	Status	Result	Check Name
> ((data_t*)&(delay_line) +* long long'(uint_type'(rptr)))	array_index	hold		
- ((data_t*)&(delay_line) +* long long'(uint_type'(wptr)))	array_index	some fail		
array_index_check_79	array_index	fail (1)	out of index bounds	array_index_check_79
array_index_check_80	array_index	hold	within index bounds	array_index_check_80
array_index_check_81	array_index	hold	within index bounds	array_index_check_81

Array index violation - Debug

Autochecks categories

The screenshot shows an IDE window with the following code:

```
62 // Read sample
63 data_t din_reg = din.read();
64
65 sc_uint<clog2<TAPS>::value> rptr = wptr;
66 mac_t mac = 0;
67 delay_line[wptr] = din_reg;
```

The error message indicates an array index violation: `0->1a 27->26`. Below the code is a logic analyzer waveform showing signals: `clk`, `rst`, `type'(wptr)`, `delay_line`, and `wptr`. The `delay_line` signal shows a failure at time `t##-1` and a hold at `t##0`. The `wptr` signal shows a value of 27 at `t##-3` and 26 at `t##0`.

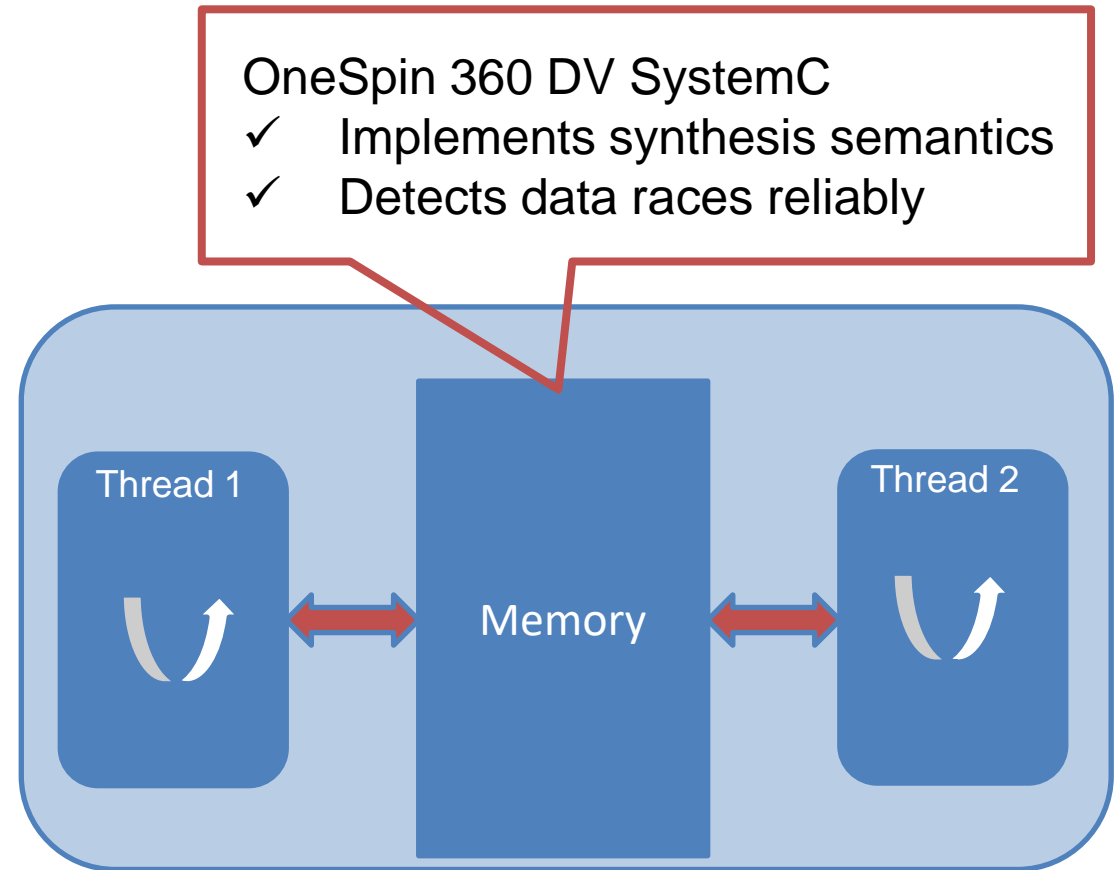
- Reason of the violation: In this case, it is signal a "delay_line". Double-click on this signal to be pointed to its declaration (it is an array of 5 elements). Go back to where "delay_line" is assigned. Notice that it contains index "wptr". If you double-click on this index and follow its driver, you will be pointed to its reset value (in this case wptr=TAPS)

```
54 wptr = (TAPS); //bug1, should be TAPS-1, creates array index violation
    27->26 27
```

SystemC Race Conditions

Simulation vs. Synthesis Mismatch

- SystemC simulation is sequential
 - Standard forces simulators to execute threads sequentially
 - No HDL-style “non-blocking” assignment
- Hardware is concurrent
 - RTL processes work in parallel
 - Synthesis result is parallel
- HLS requires careful management of concurrent access to shared memories



Write/Write races

Model Building category

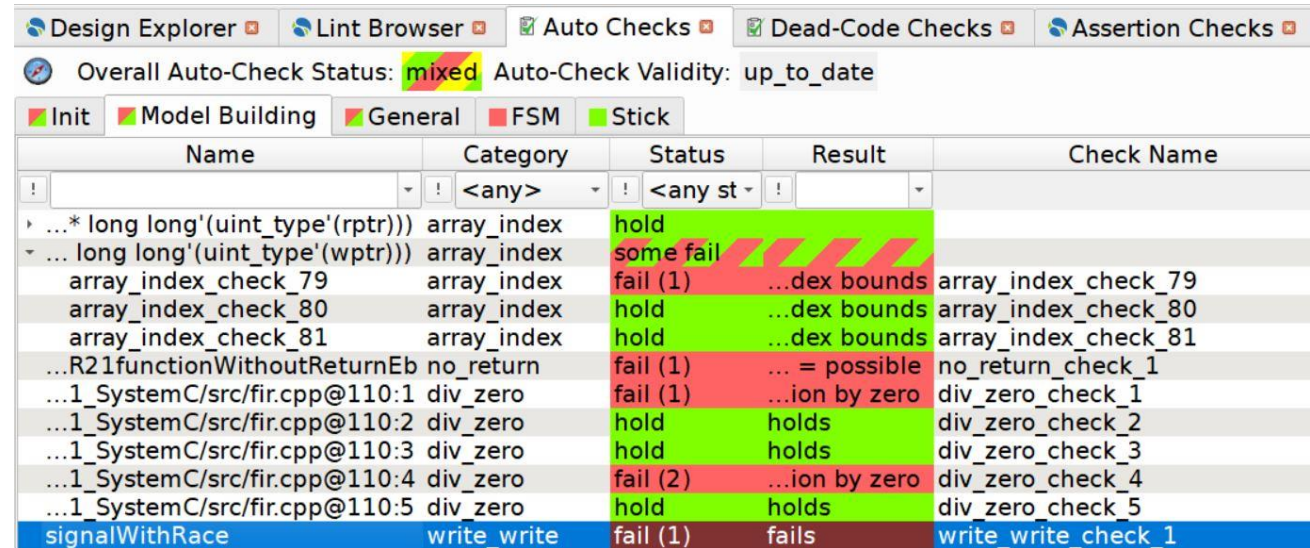
- A) **Description:** In SystemC designs, it is possible that write-write races occur among different processes. For all possibly affected signals, a write-write race check is generated, investigating whether such incident can happen.

- B) **Command to execute only these checks:**
`check_consistency -category write_write`

- C) **Example of the code:**

```
void main () {  
    signalWithRace[0]=true;  
}  
void top () {  
    signalWithRace[0]=false;  
}
```

- D) **Debug:** Debugging is done by clicking on Status field “fail” or in this example: “fail (1)” field. Counter-example waveform with active code viewer will be opened in the new window. Example on the next slide
- E) **Reason to fix:** These checks can identify unintended races between processes and incorrect design behavior. Also, it can have an affect on simulation – synthesis mismatches.



The screenshot shows the Lint Browser interface with a table of linting results. The table has columns for Name, Category, Status, Result, and Check Name. The 'signalWithRace' check is highlighted in blue, showing a 'fail (1)' status and 'fails' result.

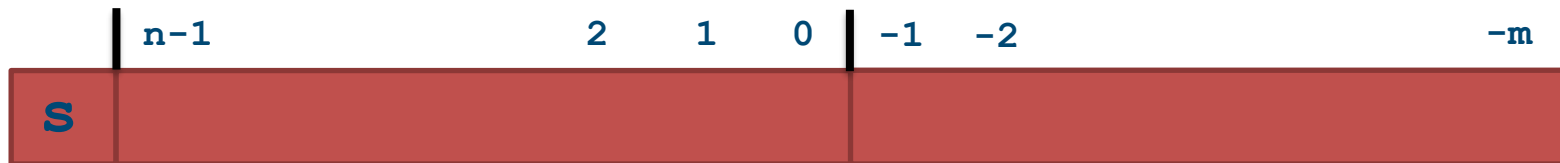
Name	Category	Status	Result	Check Name
! <any>	! <any st >	!	!	
... * long long'(uint_type'(rptr))	array_index	hold		
... long long'(uint_type'(wptr))	array_index	some fail		
array_index_check_79	array_index	fail (1)	...dex bounds	array_index_check_79
array_index_check_80	array_index	hold	...dex bounds	array_index_check_80
array_index_check_81	array_index	hold	...dex bounds	array_index_check_81
...R21functionWithoutReturnEb	no_return	fail (1)	... = possible	no_return_check_1
...1_SystemC/src/fir.cpp@110:1	div_zero	fail (1)	...ion by zero	div_zero_check_1
...1_SystemC/src/fir.cpp@110:2	div_zero	hold	holds	div_zero_check_2
...1_SystemC/src/fir.cpp@110:3	div_zero	hold	holds	div_zero_check_3
...1_SystemC/src/fir.cpp@110:4	div_zero	fail (2)	...ion by zero	div_zero_check_4
...1_SystemC/src/fir.cpp@110:5	div_zero	hold	holds	div_zero_check_5
signalWithRace	write_write	fail (1)	fails	write_write_check_1

Fixed Point Arithmetic in SystemC

Hard to Get The Precision Right in Complex Datapath

Data types `sc_(u)fixed`

- Sign + n -bit binary value
(like signed Verilog types)
- Additional m bits binary fraction
- Bit value: $a[i] * 2^i$
[fractional bits: 0.5, 0.25,...]



Advantages

- Easy to write compact arithmetic
- Implementation complexity hidden from user

Fully template-based classes

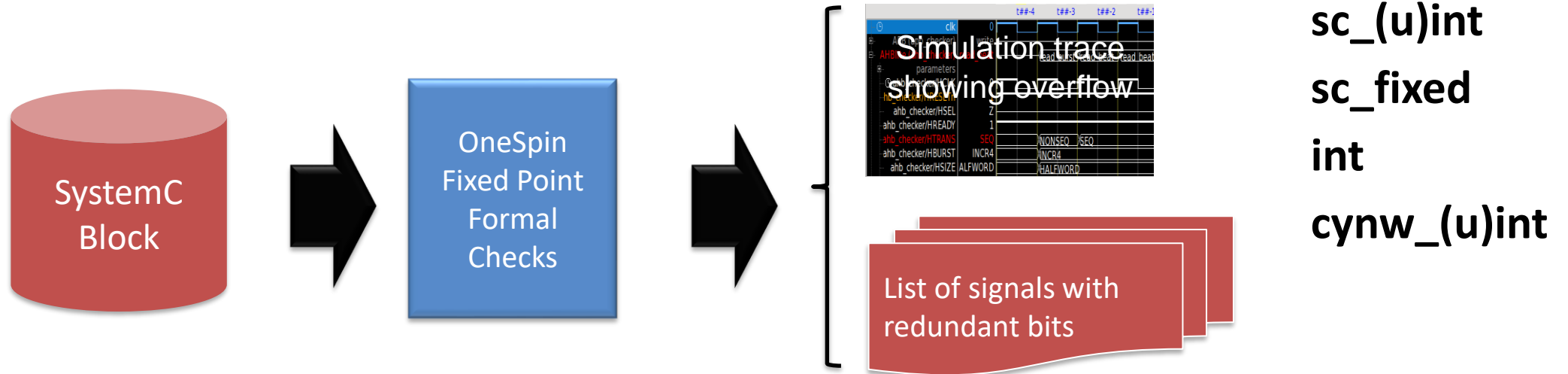
- Overloaded arithmetic operators, casts, constructors
- Can perform arithmetic on operands with different # of digits before/after decimal point

However

- Problematic to find “right” bit widths
 - Too many bits: unnecessary complexity
 - Too few bits: overflows and functional errors
- Hard to determine bit widths using simulation
 - Too many possible combinations

Fixed Point Precision App

Automated Redundancy and Overflow Checks



Check for overflow

- Check all operations for signed/unsigned overflow
- Full automation, no need for stimulus
- Prove absence of overflows
- Show traces of overflow scenarios

Check for redundant bits

- Checks uppermost bits for redundancy
- Automated, no need for stimulus
- Reports fixed point signals with redundant bits

Integer (Redundant) checks

Autochecks categories

- **A) Description:**

Integer checks are created for each signed or unsigned signal of the current unit. An integer check tests whether there are redundant bits in the signal. For an unsigned integer, it tests the most significant bits for constant. If the topmost bits are constantly zero, they are redundant and are flagged. For a signed integer, the tool tests whether the most significant bits always equal the sign bit. If so, the redundant bits are flagged.

- **B) Command to execute only these checks:**

`check_consistency -category integer`

- **C) Example of the code:**

```
sc_uint<nbits<TAPS>::value> wptr;
```

This signal contains some leading redundant bits.

- **D) Debug:** Debugging is done by clicking "Go to Source" button. The tool points you to the line where the signal is declared.

- **E) Reason to fix:** If a signal contains redundant bits, it may save area during HLS synthesis and make SystemC code efficient if the user removes these redundant bits.

Name	Category	Status	Result	Check Name
! <any>	! <any>	! <any stal >	!	
...in/golden/src/fir.cpp@70:1	truncation	fail (1)	can overflow	truncation_check_1
...in/golden/src/fir.cpp@70:2	truncation	fail (2)	can overflow	truncation_check_2
...in/golden/src/fir.cpp@70:3	truncation	hold	never overflows	truncation_check_3
buf_cnt	integer	open	unknown	integer_check_59
coeffs	integer	some fail		
delay_line	integer	hold		
dout	integer	hold	no leading redundant bits	integer_check_28
highest	integer	hold	no leading redundant bits	integer_check_57
lowest	integer	hold	no leading redundant bits	integer_check_58
o1	integer	hold	no leading redundant bits	integer_check_29
o2	integer	hold	no leading redundant bits	integer_check_61
wptr	integer	fail	found leading redundant bits	integer_check_60

Truncation (Overflow) check

General category

- A) Description:** A truncation check tests whenever overflow can happen. If the result of an integral operation is used in a context, that does not match the self-determined size or signedness of the operation, then relevant bits may be lost. The same issue may happen if some integral value is assigned to a variable with different size or signedness. For all possibly affected expressions, a truncation check is generated, investigating whether such incident can happen.

- B) Command to execute only these checks:**
`check_consistency -category truncation`

- C) Example of the code:**

```
sc_in<sc_uint<4>> in;  
sc_uint<5> lvar;  
lvar = in.read() * 10;
```

- D) Debug:** Debugging is done by clicking on Status field “fail” or in this example: “fail (1)” field. Counter-example waveform with active code viewer will be opened in the new window. Example on the next slide
- E) Reason to fix:** If overflow happens, it can cause an incorrect value and imprecise results. Fixing it before HLS can save a lot of effort to detect issue on generated RTL code

Name	Category	Status	Result	Check Name
...s/4.1_SystemC/src/fir.cpp@68:1	truncation	hold	never overflows	truncation_check_1
...s/4.1_SystemC/src/fir.cpp@68:2	truncation	hold	never overflows	truncation_check_2
...s/4.1_SystemC/src/fir.cpp@68:3	truncation	hold	never overflows	truncation_check_3
.../4.1_SystemC/src/fir.cpp@104:1	truncation	fail (1)	can overflow	truncation_check_4
.../4.1_SystemC/src/fir.cpp@104:2	truncation	hold	never overflows	truncation_check_5
.../4.1_SystemC/src/fir.cpp@104:3	truncation	hold	never overflows	truncation_check_6
.../4.1_SystemC/src/fir.cpp@104:4	truncation	fail (2)	can overflow	truncation_check_7
.../4.1_SystemC/src/fir.cpp@104:5	truncation	hold	never overflows	truncation_check_8

Toggle Checks

- “Stuck at” checks
- Easily determines which bits are not used or not tested!

OneSpin 360 (R) - Database "default" - Design: golden Unit: default

Session Setup File Edit CC/MV EC Tools Window Help

Design Explorer Lint Browser Auto Checks Dead-Code Checks Assertion Checks

Overall Auto-Check Status: **mixed** Auto-Check Validity: up_to_date

Instance	Module	Init	Model Building	General	FSM	Stick
[top]	FIR					
Name		Status	Result	Check N		
!		!	<any status>	!		
- blk1.top.blk3.mac		hold				
blk1.top.blk3.mac[0]		hold	toggles	stick_check_302		
blk1.top.blk3.mac[1]		hold	toggles	stick_check_303		
blk1.top.blk3.mac[2]		hold	toggles	stick_check_304		
blk1.top.blk3.mac[3]		hold	toggles	stick_check_305		
blk1.top.blk3.mac[4]		hold	toggles	stick_check_306		
blk1.top.blk3.mac[5]		hold	toggles	stick_check_307		
blk1.top.blk3.mac[6]		hold	toggles	stick_check_308		
blk1.top.blk3.mac[7]		hold	toggles	stick_check_309		
blk1.top.blk3.mac[8]		hold	toggles	stick_check_310		
blk1.top.blk3.mac[9]		hold	toggles	stick_check_311		
blk1.top.blk3.mac[10]		hold	toggles	stick_check_312		
blk1.top.blk3.mac[11]		hold	toggles	stick_check_313		

313 check(s) total, 12 check(s) selected by filter

DeadCode check

Dead-Code Checks category

- A) Description:** A line of code is called dead code if it is not visited in any execution trace. Lines can be unreachable, for example, if the condition of an enclosing control structure never becomes true, thus always preventing it from being executed. For each control structure, a corresponding dead code check is generated, which checks reachability of the associated line or block of source code.

- B) Command to execute only these checks:**
`check_consistency -category dead_code`

- C) Example of the code:**

```
if (in.read() & !in.read() ) {  
    tmp = 1;  
}
```

- D) Debug:** Debugging is done by clicking on Status field “fail” or in this example: “fail / unreachable” fields. Part of the code that is not reachable will be opened in the new window. Example on the next slide
- E) Reason to fix:** Pointing to unintentional design issues prior to HLS. Helping HLS tools to improve synthesis runtime.

Source	Check Name	Status	Result	Block Type
some fail				
fir.cpp:36.1-45.1	...d_code_check_22	hold	reachable (1)	process
fir.cpp:48.1-116.1	dead_code_check_1	... dead code	reachable (1)	process
fir.cpp:50.2-57.2	dead_code_check_2	hold	reachable (1)	wait
fir.cpp:60.11-115.2	dead_code_check_3	... dead code	reachable (1)	statement
fir.cpp:61.10-86.8	dead_code_check_6	... dead code	reachable (1)	wait
fir.cpp:87.10-89.3	...d_code_check_15	hold	reachable (1)	if
fir.cpp:89.10-92.3	...d_code_check_16	fail	unreachable	else
fir.cpp:95.3-112.3	...d_code_check_17	hold	reachable (1)	wait
fir.cpp:114.3	...d_code_check_20	hold	reachable (2)	wait

Division by 0

Model Building category

- **A) Description:** Division-By-Zero checks are generated for all arithmetic divisions occurring in SystemC source code, checking whether the divisor is always different from zero.

- **B) Command to execute only these checks:**
`check_consistency -category div_zero`

- **C) Example of the code:**

```
out = tmp / in.read();
```

- **D) Debug:** Debugging is done by clicking on Status field “fail” or in this example: “fail (1)” field. Counter-example waveform with active code viewer will be opened in the new window. Example on the next slide
- **E) Reason to fix:** These checks can identify unintended division by zero and by fixing them will prevent potential ‘X’ values to be propagated to output.

Name	Category	Status	Result	Check Name
! <any>	! <any st >	!	!	
...* long long'(uint_type'(rptr))	array_index	hold		
... long long'(uint_type'(wptr))	array_index	some fail		
array_index_check_79	array_index	fail (1)	...t of index bounds	array_index_check_79
array_index_check_80	array_index	hold	...hin index bounds	array_index_check_80
array_index_check_81	array_index	hold	...hin index bounds	array_index_check_81
...R21functionWithoutReturnEb	no_return	fail (1)	... result = possible	no_return_check_1
...1_SystemC/src/fir.cpp@110:1	div_zero	fail (1)	division by zero	div_zero_check_1
...1_SystemC/src/fir.cpp@110:2	div_zero	hold	holds	div_zero_check_2
...1_SystemC/src/fir.cpp@110:3	div_zero	hold	holds	div_zero_check_3
...1_SystemC/src/fir.cpp@110:4	div_zero	fail (2)	division by zero	div_zero_check_4
...1_SystemC/src/fir.cpp@110:5	div_zero	hold	holds	div_zero_check_5
signalWithRace	write_write	fail (1)	fails	write_write_check_1

Function without return

Model Building category

- **A) Description:** Function-Without-Return checks test whether each possible control path through a function ends with a return statement. With SystemC standard it is possible to have function without return, but that can cause undefined return value and have wrong influence on the design functionality

- **B) Command to execute only these checks:**
`check_consistency -category no_return`
- **C) Example of the code:**

```
int main () {  
    if (tmp)  
        out = 1;  
}
```

- **D) Debug:** Debugging is done by clicking on Status field “fail” or in this example: “fail (1)” field. Counter-example waveform with active code viewer will be opened in the new window. Example on the next slide
- **E) Reason to fix:** These checks can identify unintended function without return and fixing them can prevent unknown values that can be assigned during function call

Name	Category	Status	Result	Check Name
... * long long'(uint_type'(rptr))	array_index	hold		
... long long'(uint_type'(wptr))	array_index	some fail		
array_index_check_79	array_index	fail (1)	out of index bounds	array_index_check_79
array_index_check_80	array_index	hold	within index bounds	array_index_check_80
array_index_check_81	array_index	hold	within index bounds	array_index_check_81
...R21functionWithoutReturnEb	no_return	fail (1)	exit without result = possible	no_return_check_1
...1_SystemC/src/fir.cpp@110:1	div_zero	fail (1)	division by zero	div_zero_check_1
...1_SystemC/src/fir.cpp@110:2	div_zero	hold	holds	div_zero_check_2
...1_SystemC/src/fir.cpp@110:3	div_zero	hold	holds	div_zero_check_3
...1_SystemC/src/fir.cpp@110:4	div_zero	fail (2)	division by zero	div_zero_check_4
...1_SystemC/src/fir.cpp@110:5	div_zero	hold	holds	div_zero_check_5
signalWithRace	write_write	fail (1)	fails	write_write_check_1

Resolution X Checks

- GUI ID: `resolution_x`
- Languages: `SystemC`
- Type: `Safety`
- Counter example: `Yes`

Checks if a resolved signal can become 'X'

- `resolution_x` checks are generated for signals in systemc (including each individual bit of it) with multiple drivers
- Implement X value in the SystemC Synthesizable Standard

Shift Checks

- GUI ID: shift
- Languages: SystemC
- Type: Safety
- Counter example: Yes
- Command: `check_consistency -category shift`

Checks if a signal can become '0' by shifting its value too many times

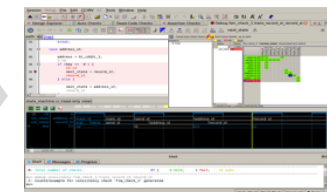
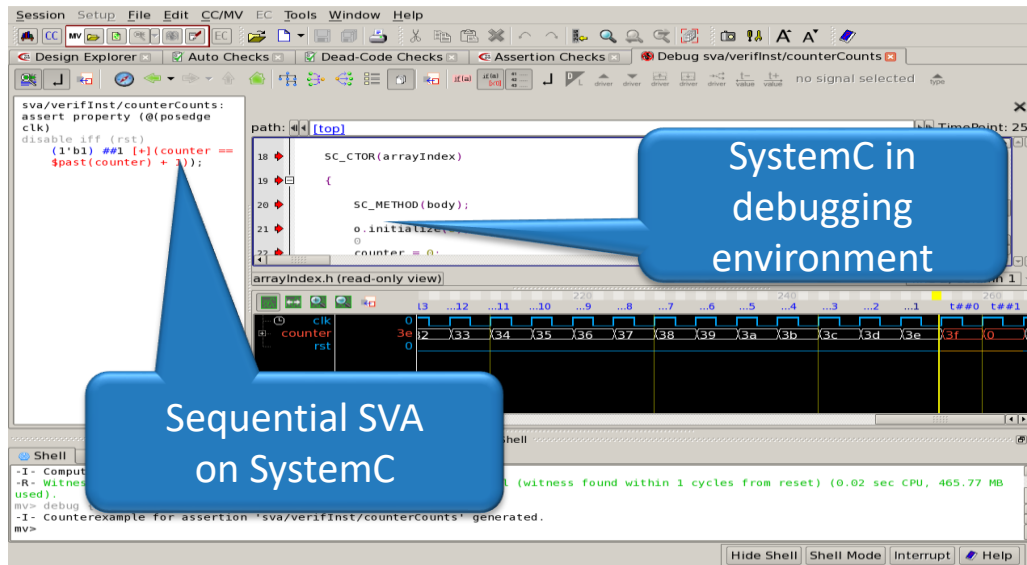
Other Capabilities



SystemC Property Checking Solution

Leveraging SVA on SystemC

- Test specification elements against algorithm
- Consistent SystemVerilog assertions pre- and post-synthesis
- Check about Specific SystemC Standard implementations



C++/SystemC Code

Formal Tool

Assertion Based Verification

Assertion Classification

Type	assert	assume	cover
Description	Assertion	Constraint	Cover point
Purpose	Monitor DUT behavior	"Monitor" DUT inputs	Collect coverage data
Simulation	Eliminate 'fail' from TBs		Achieve 'pass' in TBs
Formal	Ensure absence of 'fail' by proving assertion	Assume absence of 'fail' (never show trace where assume fails)	Automatically find 'pass' or prove absence of 'pass'

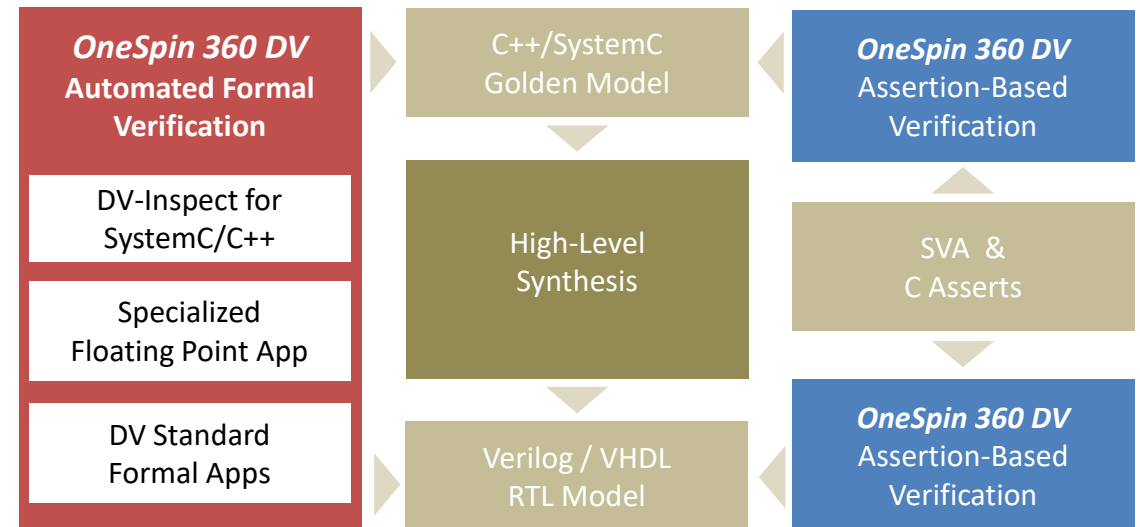
- Distinction between assert/assume only important for formal
- Formal typically requires assumes in order to avoid unrealistic fails for asserts

OneSpin with HLS Partnership

Design Verification Solution for HLS tools

- Cooperation with HLS teams
- Support of HLS libraries and coding
- Provides *independent* check on HLS flow

More efficient analysis and debug of C++/SystemC model prior high-level synthesis



Re-Use of assertions and apps on RTL for consistency

Use formal first! Improves verification flow! Gets working RTL faster!

OneSpin SystemC/C++ Solution

Enabling the HLS Flow

SystemC/C++ Hardware Verification

- Currently tools do not address verification challenges
- HLS driving need for pre-synthesis verification

Language and Algorithm Verification Needs

- SystemC artifacts cause problems downstream
- Algorithm verification can be accelerated with automation

OneSpin: Unique SystemC Formal Solution

- Automation to significantly improve SystemC testing
- SystemVerilog assertions for flow continuity

**For more information,
please visit**

www.onespin.com

Thank You!



Disclaimer

- © Siemens 2021
- Subject to changes and errors. The information given in this document only contains general descriptions and/or performance features which may not always specifically reflect those described, or which may undergo modification in the course of further development of the products. The requested performance features are binding only when they are expressly agreed upon in the concluded contract.
- All product designations may be trademarks or other rights of Siemens AG, its affiliated companies or other companies whose use by third parties for their own purposes could violate the rights of the respective owner.

Copyright Permission

- A non-exclusive, irrevocable, royalty-free copyright permission is granted by **OneSpin: A Siemens Business** to use this material in developing all future revisions and editions of the resulting draft and approved Accellera Systems Initiative **SystemC** standard, and in derivative works based on the standard.