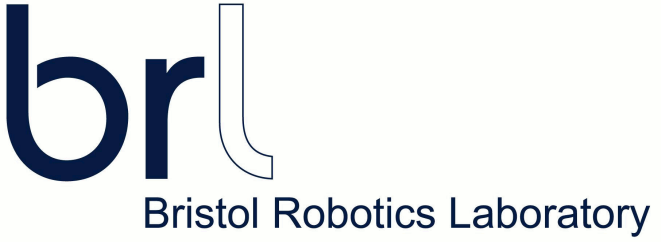
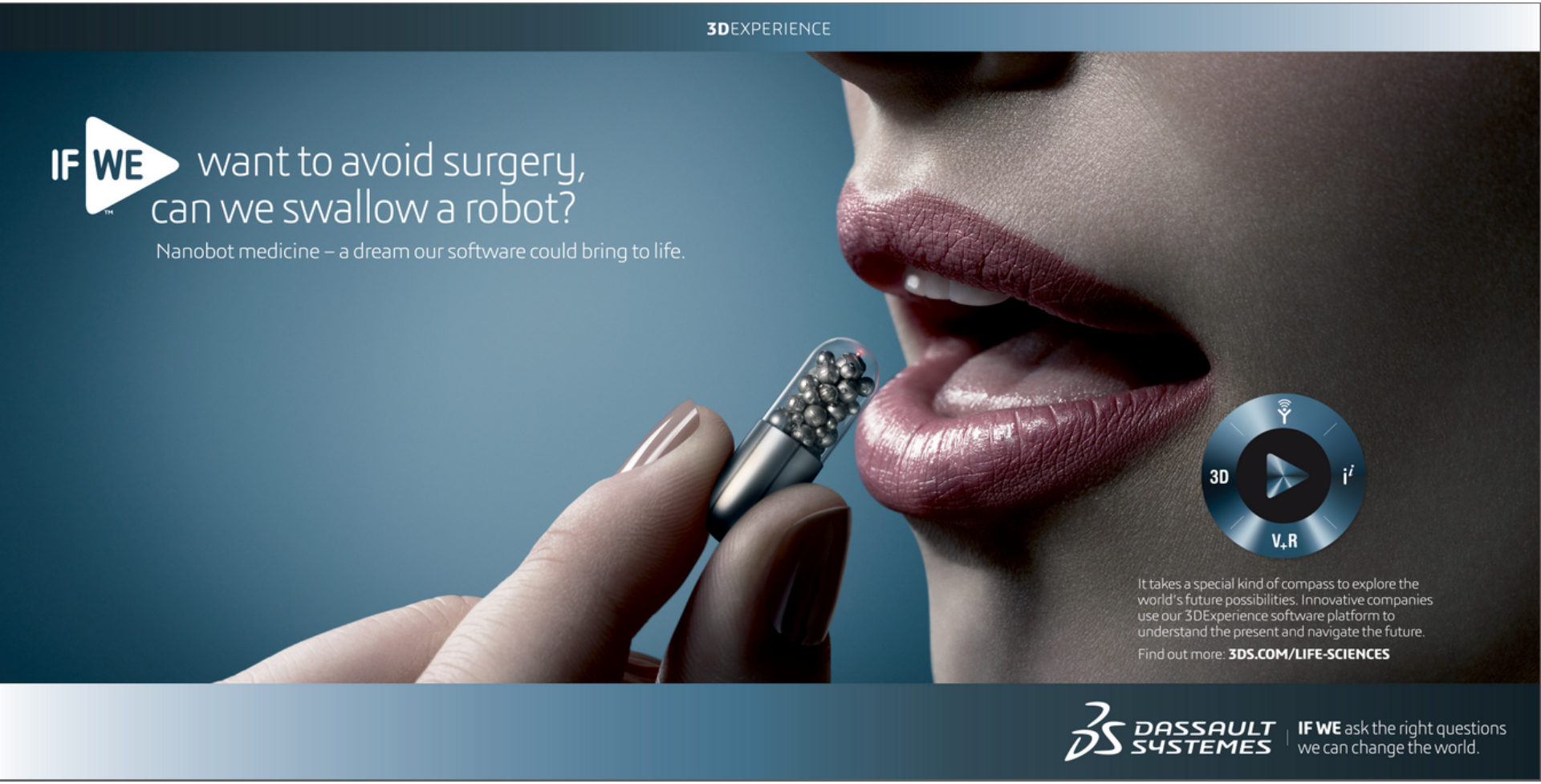
Practical Techniques for Verification and Validation of Robots

Kerstin Eder
with a demo by Dejanira Araiza Illan

University of Bristol and
Bristol Robotics Laboratory



Would you swallow a robot?



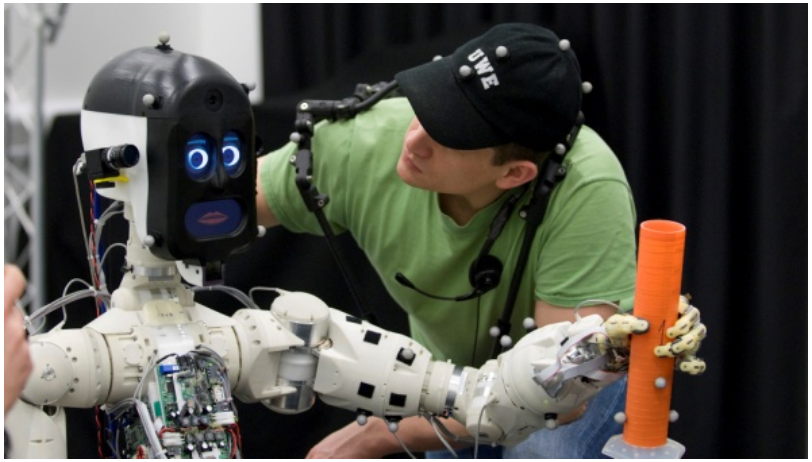
The Safety Challenge

- Autonomous Systems
- Engineering Challenge
  - Advances in control science
  - Focus on “making things work”

Pictures from www.wikipedia.org

The Safety Challenge

- Autonomous Systems
- Engineering Challenge
  - Advances in control science
  - Focus on “making things work”
- Fundamental concern:
  - Can such systems be trusted?



Dependability

- A system is dependable (or trustworthy) only if it can be **shown** to be safe and useful.
  - Safety is the property of avoiding harmful conditions.
  - Liveness requires that the system achieves its goals a.k.a. usefulness.

- **Demonstrable** safety and liveness are required.

Safety Assurance

Assurance is the essential concept:

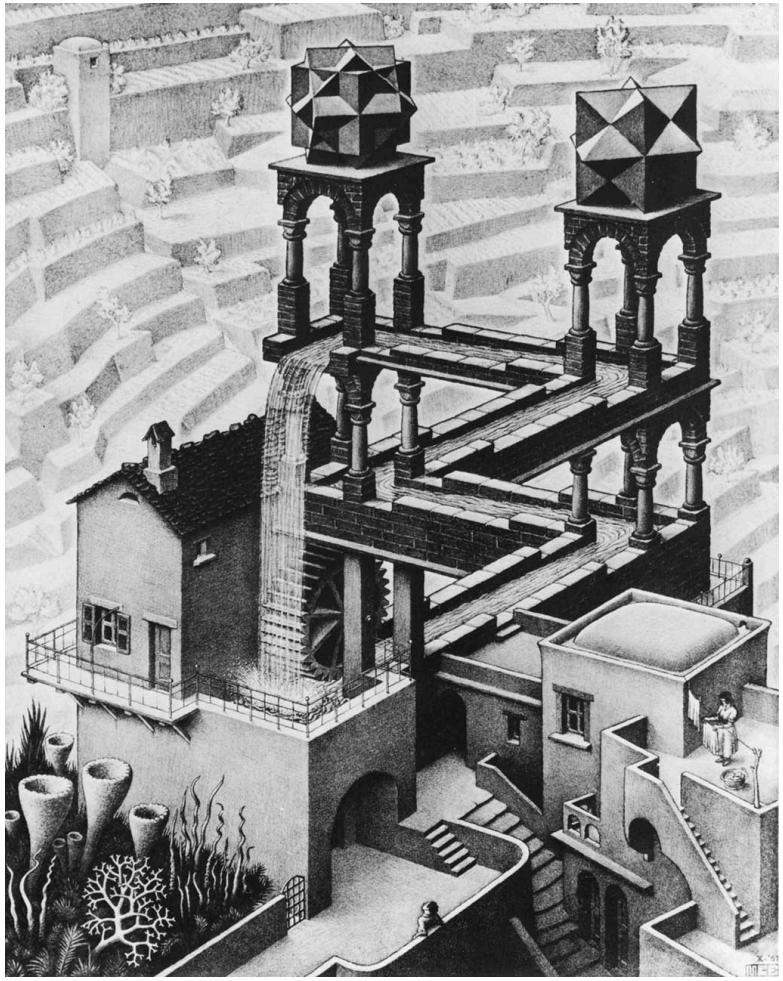
- A system may never cause harm throughout its entire operating life,
- but if we cannot be assured of that before we start to use it, then the system can not be trusted.

Designing Dependable Systems

- Create flawless designs.

AND

- Design the system in such a way that the flawlessness can be demonstrated.



"Waterfall" by M.C Escher.

EPSRC “Principles of Robotics”

“Robots are products.

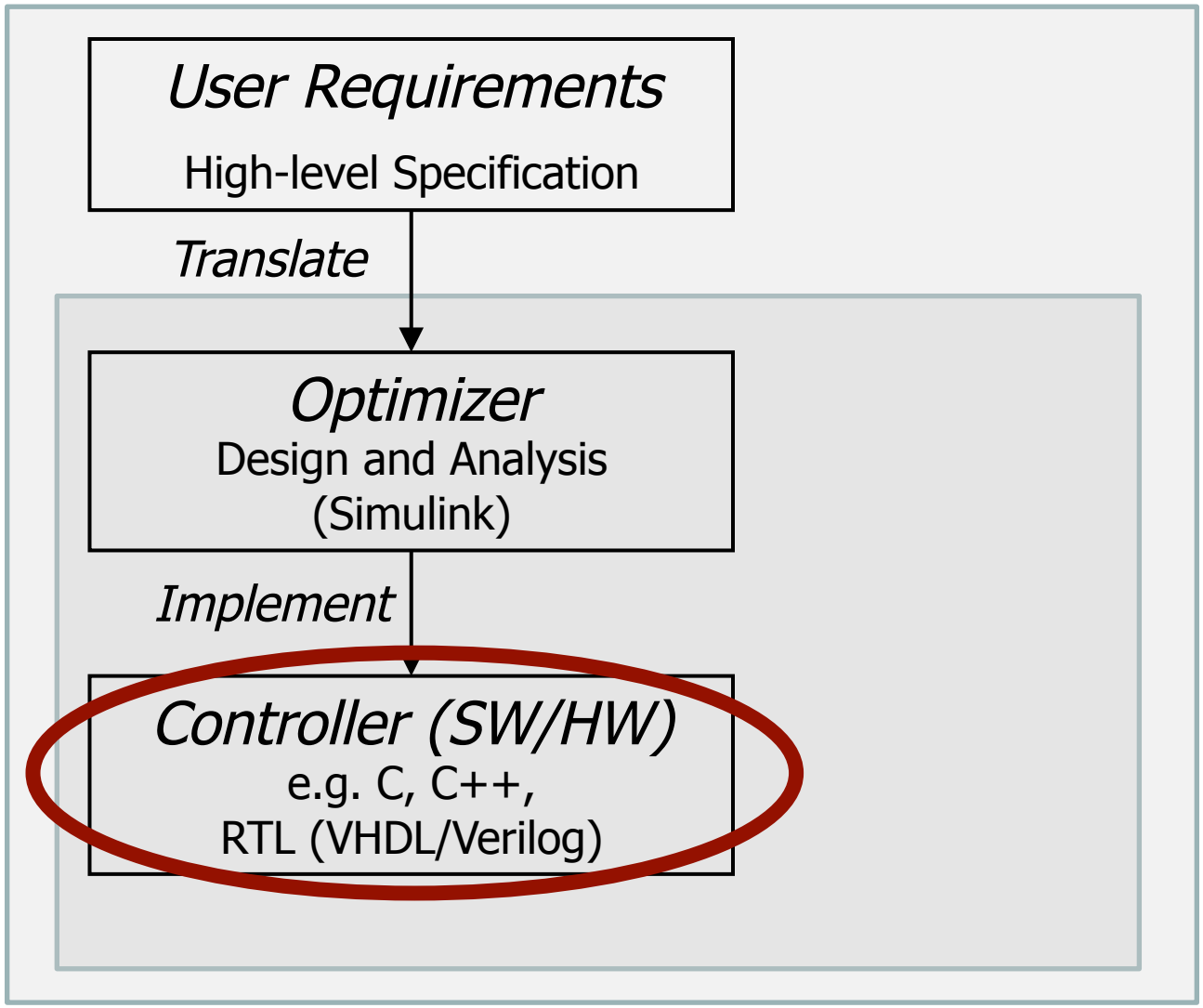
*They should be **designed** using **processes which assure** their **safety** and security.”*

http://www.epsrc.ac.uk/ourportfolio/themes/engineering/activities/Pages/principlesofrobotics.aspx

Verification and Validation for Safety in Robots

To develop techniques and methodologies that can be used to design autonomous intelligent systems that are demonstrably trustworthy.

Correctness from specification to implementation



What can be done at the code level?

P. Trojanek and K. Eder.
Verification and testing of mobile robot navigation algorithms: A case study in SPARK.
IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). pp. 1489-1494. Sep 2014.
http://dx.doi.org/10.1109/IROS.2014.6942753

What can go wrong in robot navigation software?

Generic bugs:

- Array and vector out-of-bounds accesses
- Null pointer dereferencing
- Accesses to uninitialized data

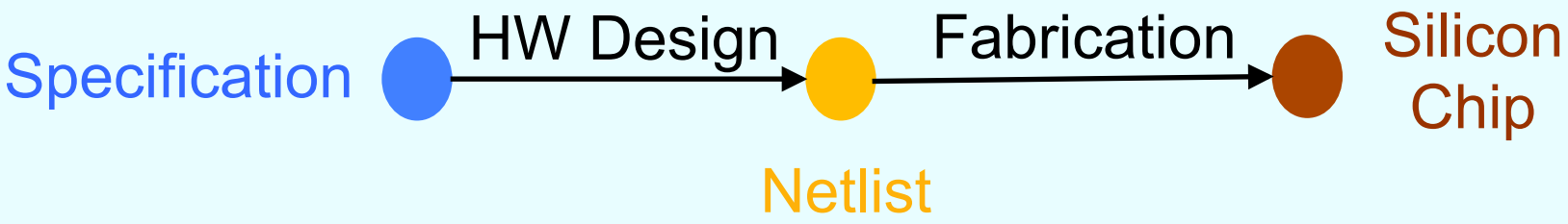
Domain-specific bugs:

- Integer and floating-point arithmetic errors
- Mathematic functions domain errors
- Dynamic memory allocation errors
- Concurrency bugs
  - blocking inter-thread communication (non real-time)

State of the art verification approaches

- Model checking
  - infeasible for real (off-the-shelf) code
- Static analysis of C++
  - not possible
- Static analysis of C
  - requires verbose and difficult to maintain annotations

HW Design Reconvergence Model



Why design a custom integrated circuit?

Mask costs versus line width

Process (µm)	Vdd	Metal	Gates/sq mm	Mask set cost ($)
0.065	1.0	9	400k	3,000,000
0.09	1.0	9	200k	1,500,000
0.13	1.2	7	100k	750,000
0.18	1.8	5	40k	250,000
0.25	2.5	5	24k	150,000
0.35	3.3	3	12k	40,000
0.5	3.3	3	5k	20,000
0.6	5.0	2	4k	18,000

Source: 'Asic Design in the Silicon Sandbox: A Complete Guide to Building Mixed-Signal Integrated Circuits'. (The McGraw-Hill Companies).

→ Furthermore, there are other energy-saving techniques you can use that are unique to custom ICs.

For some applications, small size and weight are crucial. Just open up an MP3 player, mobile phone, digital camera or laptop computer for examples of tight and light design. When a set of standard parts is too large or heavy, a custom chip is required.

A designer with access to the full flexibility of a custom chip can create numerous special functions that are difficult to find elsewhere.

For example, special purpose arithmetic units, multi-port memories, and a variety of non-volatile storage circuits can be developed. One can even create magnetic sensors and light sensors ranging from a single sensor to line sensors and two-dimensional video camera chips.

Some companies use custom ICs to better protect their intellectual property. A custom integrated circuit is much more difficult to reverse engineer than a board level design.

The benefits: reliability

Higher integration levels bring greater system reliability.

If your board, with dozens of parts and hundreds of solder connections, can be replaced by one or a few parts with fewer board-level interconnects then the system becomes more reliable. Likewise, higher integration leads to lower manufacturing costs. If the custom IC uses less power, you may be able to use a cheaper power supply. Fewer boards also mean fewer connectors and smaller, less-expensive cabinets.

One company built a product that had two discrete transistors, a photocell, and a few resistors and capacitors. The circuit board was larger than they needed, they had a measurable field failure rate, and it cost about $1.00.

The company designed a custom IC with several thousand transistors to implement the same function. It had no measurable field failure rate, and the unit cost was about $0.50. For the millions of units sold, the payback on this custom chip investment was rapid.

The downside: cost

Custom chips have higher tooling costs, so if it is important to minimise the cost of prototypes by using standard parts. You may be able to gather a few PCBs and a handful of parts, hand-solder them together, and demonstrate a prototype for about $2,000. The tooling costs of a custom IC start at about $18,000 for a set of masks for a 0.6µm process and go up to about $3m for a 65nm process.

Products that have high volumes and require huge amounts of processing and memory will need the finest line width processes to get the lowest cost in production. However, for most other products, the manufacturing volumes never make sense for the $3m tooling cost. Fortunately, the tooling for coarser line widths is much more affordable, yet still larger than that of a PCB.

The downside: time

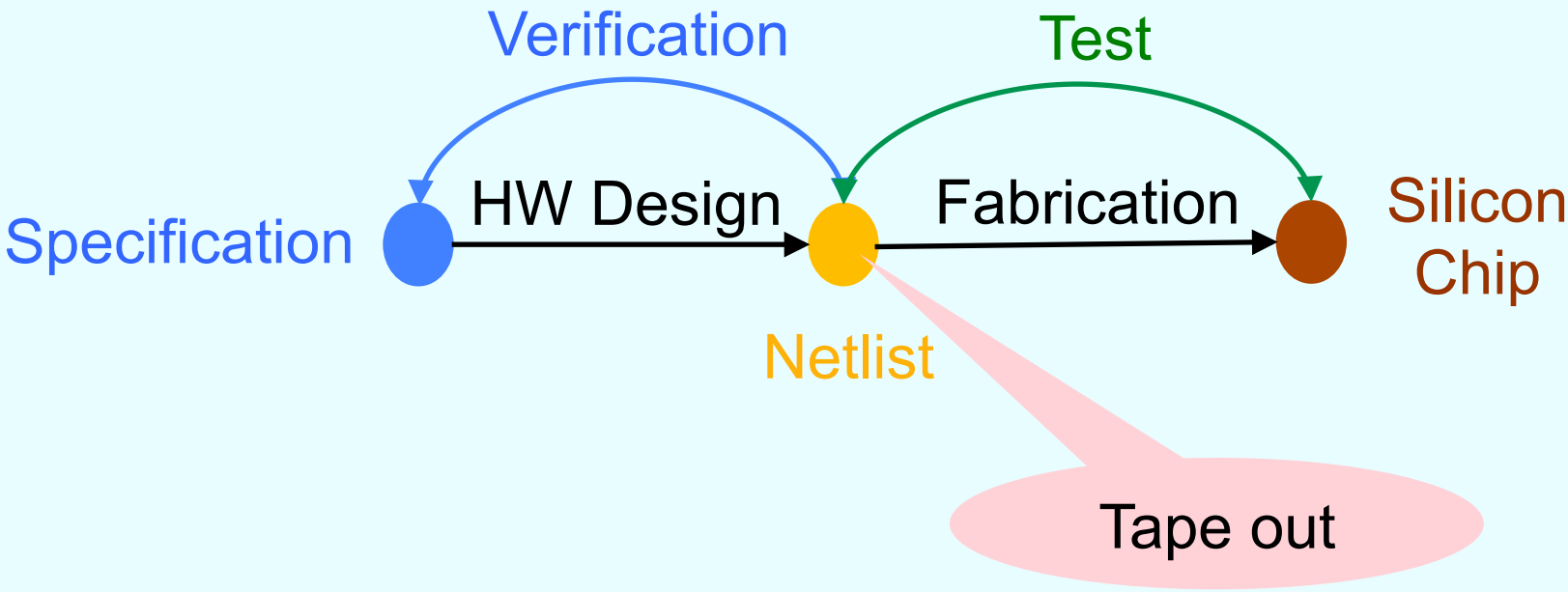
Custom chips also have longer turnaround times.

If you have a good relationship with your board supplier, a board can be manufactured in a couple of days. Add some shipping and assembly time, and you will still get a new prototype built using standard parts in less than a week.

If you go custom IC, it will be weeks, if not months, before the first chips arrive at your door. And although expediting is often available, the fees are steep and shave only a few days off a lengthy →p30 process.

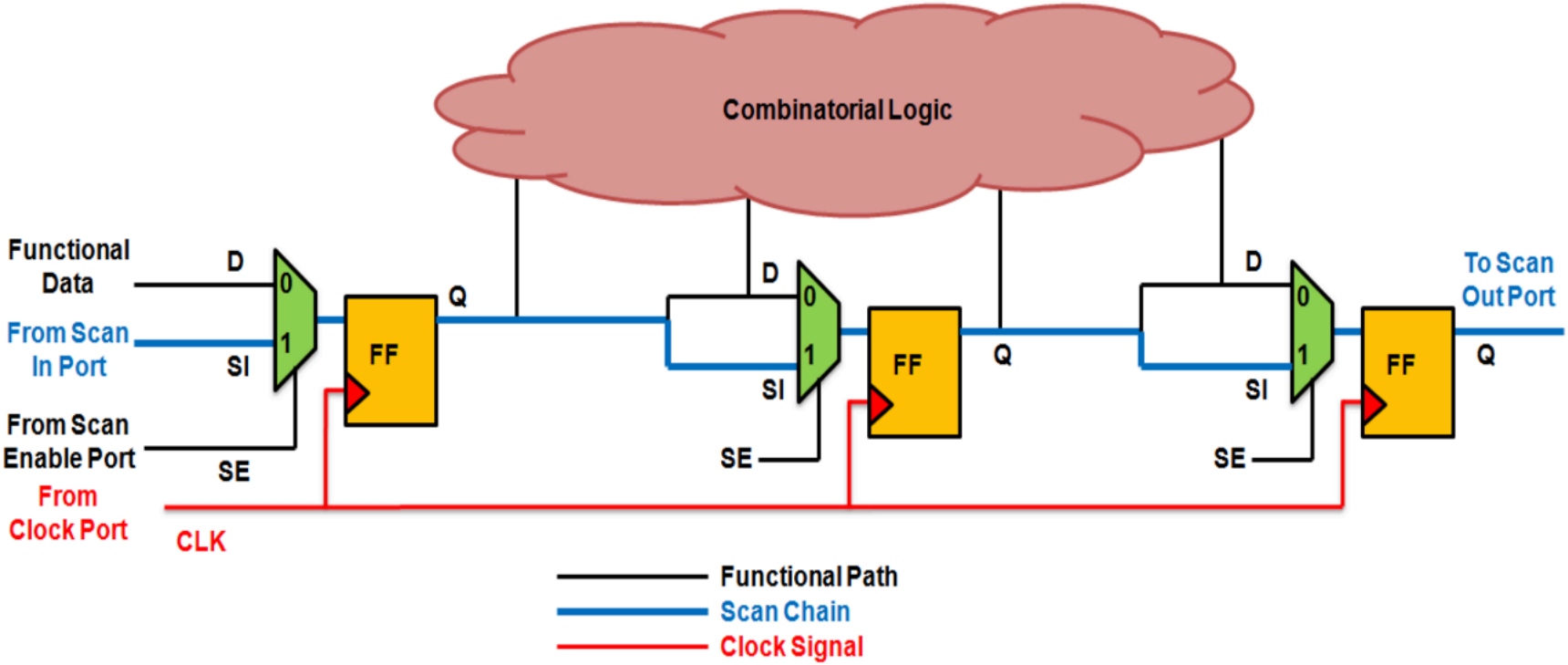
If you go custom IC, it will be weeks, if not months, before the first chips arrive at your door. And although expediting is often available, the fees are steep and shave only a few days off a lengthy process

HW Design Reconvergence Model



- ***Design Verification*** *is the process used to gain confidence in the correctness of a design w.r.t. the requirements and specification.*
- ***Test:*** Manufacturing Test vs Functional Test

Design-for-Test(ability)



http://anysilicon.com/overview-and-dynamics-of-scan-testing/

Design-for-Verification Approach

- **SPARK,** a verifiable subset of Ada
  - software reliability a primary goal
  - no memory allocation, pointers, concurrency
  - side-effect free functions
  - SPARK specification and tools freely available for academic use
- Required code modifications:
  - Pre- and post-conditions, loop (in)variants
  - Numeric subtypes (e.g. positive)
  - Formal data containers

Navigation in SPARK

- Three open-source implementations of navigation algorithms translated from C/C++ (2.7 kSLOC) to SPARK (3.5 kSLOC)
  - Vector Field Histogram
  - Nearness Diagram
  - Smooth Nearness-Diagram

	Driver	**Algorithm**	
	C++	C/C++	Ada
VFH+	807	782	918
ND	828	1037	1426
SND	403	941	1183
Total	2038	2760	3527

- Explicit annotations are less than 5% of the code
- SPARK code is on average 30% longer than C/C++

Verification Conditions

	Explicit annotations					Implicit run-time checks						Total
	Pre-conditions*	Post-conditions	Loop invariants*	Loop variants	Assertions	Divisions	Integer overflows	Floating-point overflows	Subtype ranges	Array indices	Record discriminants	
VFH+	46 (3)	5	18 (9)	0	23	36	36	120	100	102	262	**748**
ND	83 (18)	10	8 (4)	2	3	54	23	254	53	50	0	**540**
SND	104 (9)	9	14 (7)	2	30	29	1	140	22	0	24	**375**

* Separate verification conditions are generated for each call to subprogram with precondition, and similarly for initialization and preservation of each loop invariant; the numbers of explicit annotations are given in brackets.

Formal Verification Results

	Alt-Ergo 0.96	Z3 4.3.1	Alt-Ergo & Z3 combined	**Total**
VFH+	633 11 min	699 37 min	701 48 min	**748**
ND	462 17 min	482 21 min	483 41 min	**540**
SND	350 29 min	366 6 min	366 36 min	**375**

Number of discharged verification conditions and the running time of static analysis

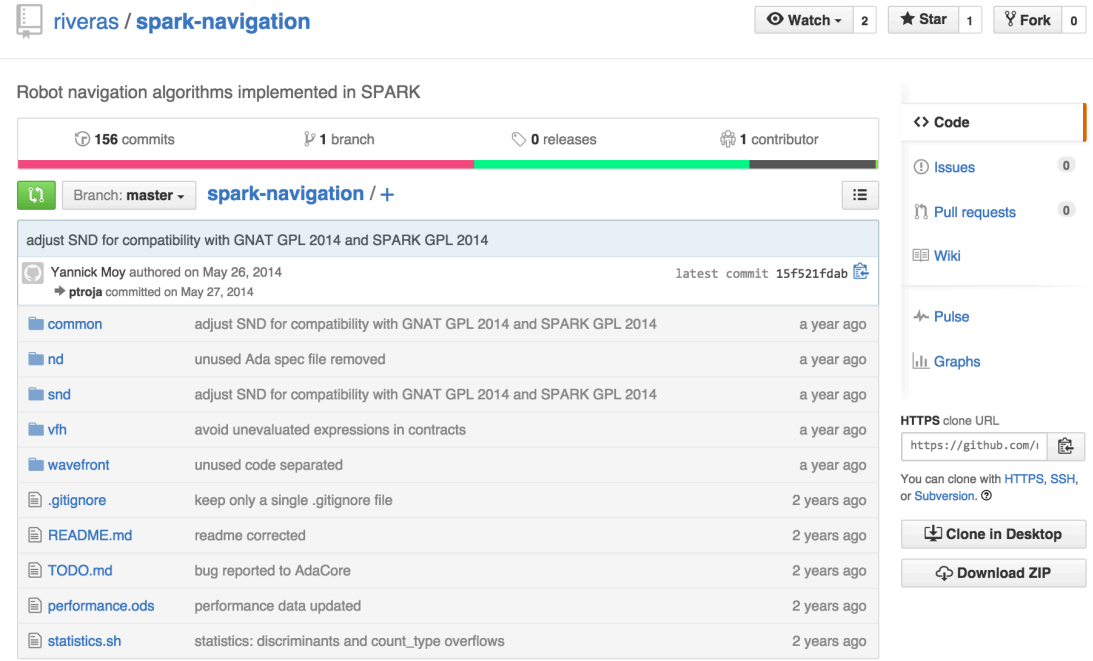
Results

- Several bugs discovered by run-time checks injected by the Ada compiler
  - Fixed code proved to be run-time safe
    - except floating-point over- and underflows
    - These require the use of complementary techniques, e.g. abstract interpretation.
- Up to 97% of the verification conditions discharged automatically by SMT solvers in less than 10 minutes
- Performance of the SPARK and C/C++ code similar

Moral: If you want to make runtime errors an issue of the past, then select your tools (programming language and dev env) wisely!

Moral

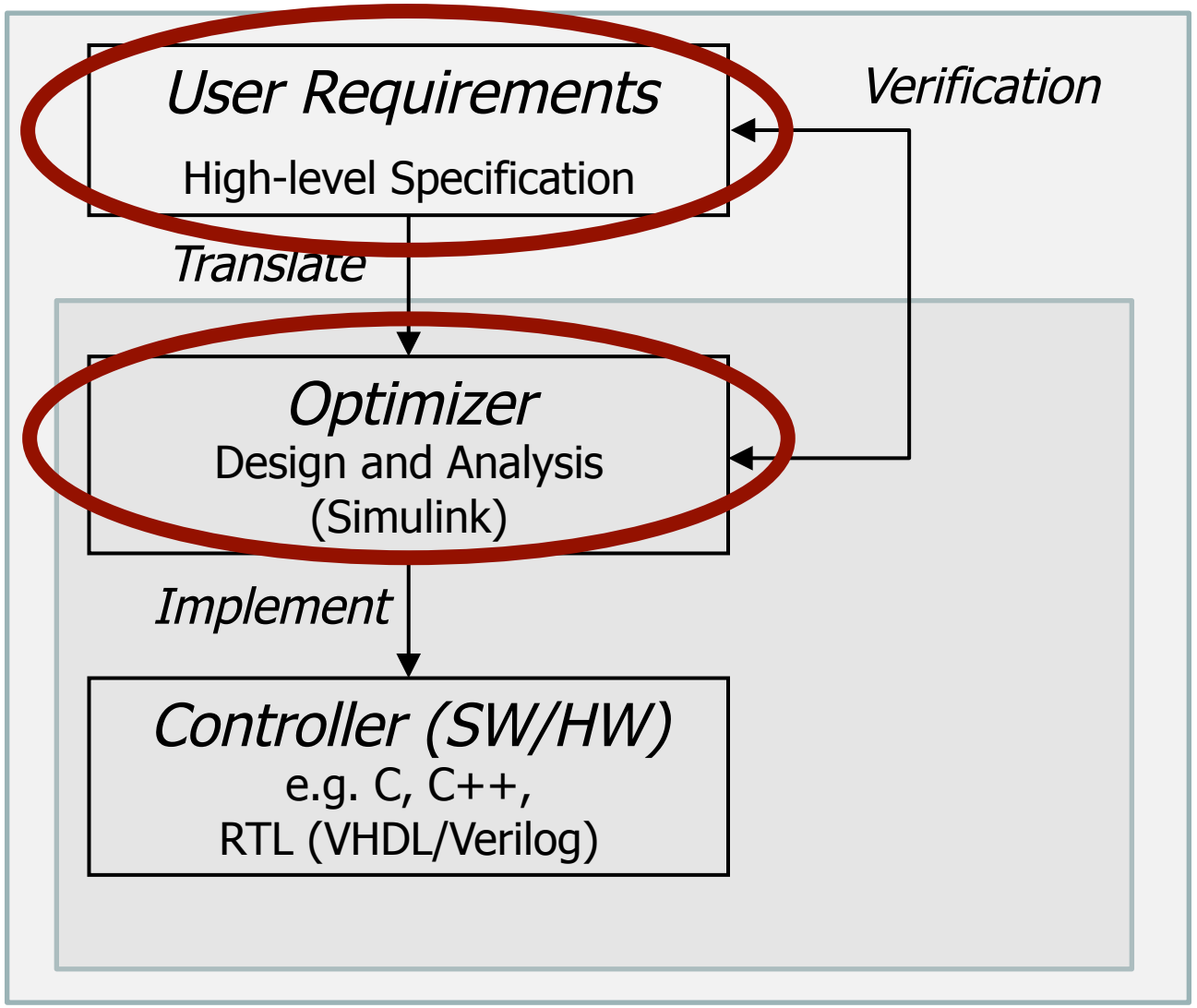
If you want to make runtime errors an issue of the past, then you must select your tools (programming language and dev env) wisely!



http://github.com/riveras/spark-navigation

P. Trojanek and K. Eder.
Verification and testing of mobile robot navigation algorithms: A case study in SPARK.
IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). pp. 1489-1494. Sep 2014.
http://dx.doi.org/10.1109/IROS.2014.6942753

Correctness from Specification to Implementation



What can be done at the design level?

D. Araiza Illan, K. Eder, A. Richards.
Formal Verification of Control Systems' Properties with Theorem Proving.
International Conference on Control (CONTROL), pp. 244 – 249. IEEE, Jul 2014.
http://dx.doi.org/10.1109/CONTROL.2014.6915147

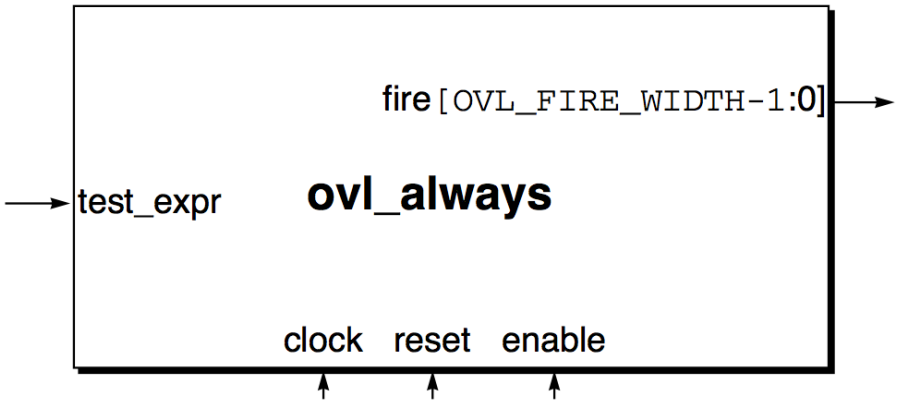
D. Araiza Illan, K. Eder, A. Richards.
***Verification of Control Systems Implemented in Simulink with Assertion Checks and Theorem Proving: A Case Study*.**
European Control Conference (ECC), pp. tbc. Jul 2015.
http://arxiv.org/abs/1505.05699

What is an assertion?

- An assertion is a statement that a particular property is required to be true.
  - A property is a Boolean-valued expression
- Assertions can be checked either during simulation or using a formal property checker.
- Assertions have been used in SW design for a long time.
  - `assert()` function is part of C `#include <assert.h>`
  - Used to detect `NULL` pointers, out-of-range data, ensure loop invariants, etc.
- Revolution through **Foster & Bening's OVL for Verilog**.
  - Clever way of encoding re-usable assertion library in Verilog. ☺
  - > 30 checker types (assertion templates)
  - http://accellera.org/activities/working-groups/ovl

ovl_always

Checks that the value of an expression is TRUE.



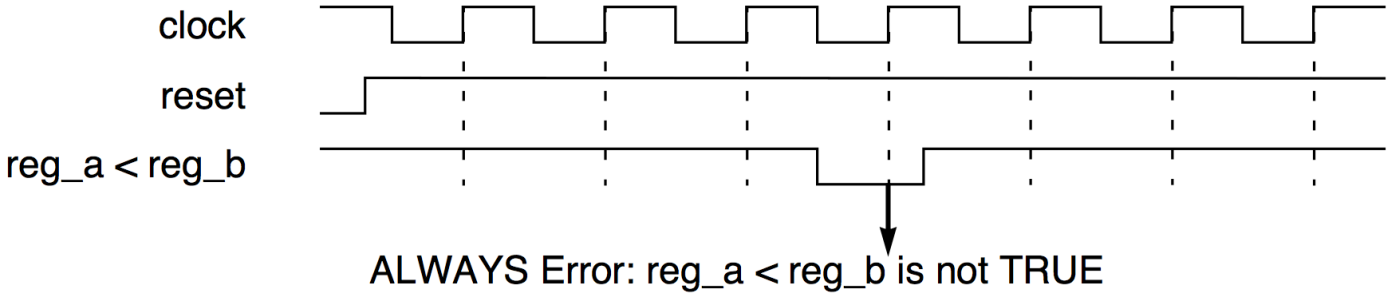
Parameters/Generics:
severity_level
property_type
msg
coverage_level
clock_edge
reset_polarity
gating_type

Class: 1-cycle assertion

Syntax

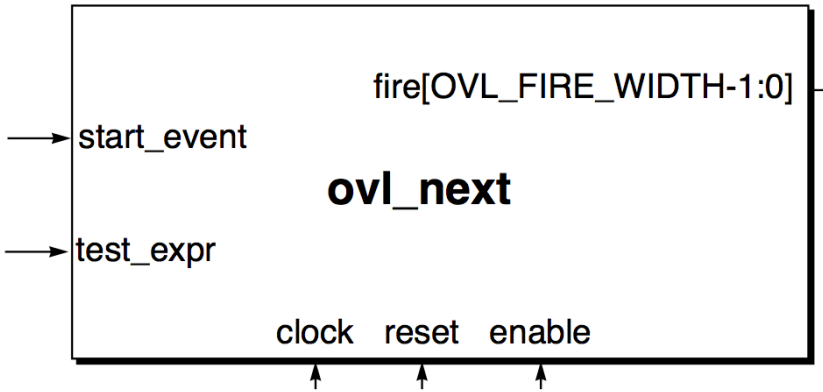
```
ovl_always
      [#(severity_level, property_type, msg, coverage_level, clock_edge,
         reset_polarity, gating_type)]
   instance_name (clock, reset, enable, test_expr, fire);
```

Checks that (*reg_a* < *reg_b*) is TRUE at each rising edge of *clock*.



ovl_next

Checks that the value of an expression is TRUE a specified number of cycles after a start event.



Parameters/Generics:

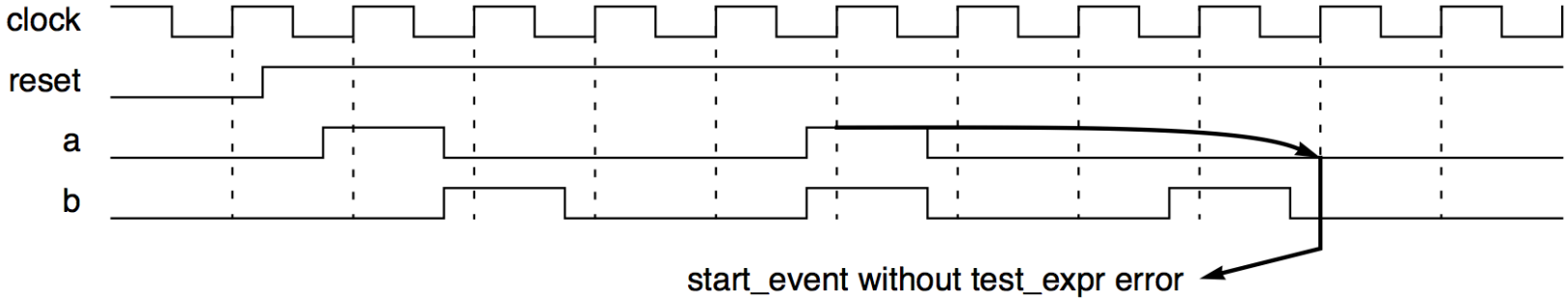
severity_level	*msg*
num_cks	*coverage_level*
check_overlapping	*clock_edge*
check_missing_start	*reset_polarity*
property_type	*gating_type*

Class: *n*-cycle assertion

Syntax

```
ovl_next
      [#(severity_level, num_cks, check_overlapping, check_missing_start,
         property_type, msg, coverage_level, clock_edge, reset_polarity,
         gating_type)]
   instance_name (clock, reset, enable, start_event, test_expr, fire);
```

Checks that *b* is TRUE 4 cycles after *a* is TRUE.



OVL QUICK REFERENCE (www.eda.org/ovl) Last updated: 28th April 2006

TYPE	NAME	PARAMETERS	PORTS	DESCRIPTION
Single-Cycle	**assert_always**	#(severity_level, property_type, msg, coverage_level)	(clk, reset_n, **test_expr**)	test_expr must always hold
Two Cycles	**assert_always_on_edge**	#(severity_level, **edge_type**, property_type, msg, coverage_level)	(clk, reset_n, **sampling_event**, **test_expr**)	test_expr is true immediately following the specified edge (edge_type: 0=no-edge, 1=pos, 2=neg, 3=any)
n-Cycles	**assert_change**	#(severity_level, **width**, **num_cks**, **action_on_new_start**, property_type, msg, coverage_level)	(clk, reset_n, **start_event**, **test_expr**)	test_expr must change within num_cks of start_event (action_on_new_start: 0=ignore, 1=restart, 2=error)
n-Cycles	**assert_cycle_sequence**	#(severity_level, **num_cks**, **necessary_condition**, property_type, msg, coverage_level)	(clk, reset_n, **event_sequence**)	if the initial sequence holds, the final sequence must also hold (necessary_condition: 0=trigger-on-most, 1=trigger-on-first, 2=trigger-on-first-unpipelined)
Two Cycles	**assert_decrement**	#(severity_level, **width**, **value**, property_type, msg, coverage_level)	(clk, reset_n, **test_expr**)	if test_expr changes, it must decrement by the value parameter (modulo 2^width)
Two Cycles	**assert_delta**	#(severity_level, **width**, **min**, **max**, property_type, msg, coverage_level)	(clk, reset_n, **test_expr**)	if test_expr changes, the delta must be >=min and <=max
Single Cycle	**assert_even_parity**	#(severity_level, **width**, property_type, msg, coverage_level)	(clk, reset_n, **test_expr**)	test_expr must have an even parity, i.e. an even number of bits asserted
Two Cycles	**assert_fifo_index**	#(severity_level, **depth**, **push_width**, **pop_width**, property_type, msg, coverage_level, **simultaneous_push_pop**)	(clk, reset_n, **push**, **pop**)	FIFO pointers should never overflow or underflow
n-Cycles	**assert_frame**	#(severity_level, **min_cks**, **max_cks**, **action_on_new_start**, property_type, msg, coverage_level)	(clk, reset_n, **start_event**, **test_expr**)	test_expr must not hold before min_cks cycles, but must hold at least once by max_cks cycles (action_on_new_start: 0=ignore, 1=restart, 2=error)
n-Cycles	**assert_handshake**	#(severity_level, **min_ack_cycle**, **max_ack_cycle**, **req_drop**, **deassert_count**, **max_ack_length**, property_type, msg, coverage_level)	(clk, reset_n, **req**, **ack**)	req and ack must follow the specified handshaking protocol
Single-Cycle	**assert_implication**	#(severity_level, property_type, msg, coverage_level)	(clk, reset_n, **antecedent_expr**, **consequent_expr**)	if antecedent_expr holds then consequent_expr must hold in the same cyle
Two Cycles	**assert_increment**	#(severity_level, **width**, **value**, property_type, msg, coverage_level)	(clk, reset_n, **test_expr**)	if test_expr changes, it must increment by the value parameter (modulo 2^width)
Single-Cycle	**assert_never**	#(severity_level, property_type, msg, coverage_level)	(clk, reset_n, **test_expr**)	test_expr must never hold
Single-Cycle	**assert_never_unknown**	#(severity_level, **width**, property_type, msg, coverage_level)	(clk, reset_n, **qualifier**, **test_expr**)	test_expr must never be an unknown value, just boolean 0 or 1
Combinatorial	**assert_never_unknown_async**	#(severity_level, **width**, property_type, msg, coverage_level)	(reset_n, **test_expr**)	test_expr must never go to an unknown value asynchronously, it must remain boolean 0 or 1
n-Cycles	**assert_next**	#(severity_level, **num_cks**, **check_overlapping**, **check_missing_start**, property_type, msg, coverage_level)	(clk, reset_n, **start_event**, **test_expr**)	test_expr must hold num_cks cycles after start_event holds
Two Cycles	**assert_no_overflow**	#(severity_level, **width**, **min**, **max**, property_type, msg, coverage_level)	(clk, reset_n, **test_expr**)	if test_expr is at max, in the next cycle test_expr must be >min and <=max
Two Cycles	**assert_no_transition**	#(severity_level, **width**, property_type, msg, coverage_level)	(clk, reset_n, **test_expr**, **start_state**, **next_state**)	if test_expr==start_state, in the next cycle test_expr must not change to next_state
Two Cycles	**assert_no_underflow**	#(severity_level, **width**, **min**, **max**, property_type, msg, coverage_level)	(clk, reset_n, **test_expr**)	if test_expr is at min, in the next cycle test_expr must be >=min and <max
Single-Cycle	**assert_odd_parity**	#(severity_level, **width**, property_type, msg, coverage_level)	(clk, reset_n, **test_expr**)	test_expr must have an odd parity, i.e. an odd number of bits asserted
Single-Cycle	**assert_one_cold**	#(severity_level, **width**, **inactive**, property_type, msg, coverage_level)	(clk, reset_n, **test_expr**)	test_expr must be one-cold i.e. exactly one bit set low (inactive: 0=also-all-zero, 1=also-all-ones, 2=pure-one-cold)
Single-Cycle	**assert_one_hot**	#(severity_level, **width**, property_type, msg, coverage_level)	(clk, reset_n, **test_expr**)	test_expr must be one-hot i.e. exactly one bit set high
Combinatorial	**assert_proposition**	#(severity_level, property_type, msg, coverage_level)	(reset_n, **test_expr**)	test_expr must hold asynchronously (not just at a clock edge)
Two Cycles	**assert_quiescent_state**	#(severity_level, **width**, property_type, msg, coverage_level)	(clk, reset_n, **state_expr**, **check_value**, **sample_event**)	state_expr must equal check_value on a rising edge of sample_event (also checked on rising edge of `OVL_END_OF_SIMULATION)
Single-Cycle	**assert_range**	#(severity_level, **width**, **min**, **max**, property_type, msg, coverage_level)	(clk, reset_n, **test_expr**)	test_expr must be >=min and <=max
n-Cycles	**assert_time**	#(severity_level, **num_cks**, **action_on_new_start**, property_type, msg, coverage_level)	(clk, reset_n, **start_event**, **test_expr**)	test_expr must hold for num_cks cycles after start_event (action_on_new_start: 0=ignore, 1=restart, 2=error)
Two Cycles	**assert_transition**	#(severity_level, **width**, property_type, msg, coverage_level)	(clk, reset_n, **test_expr**, **start_state**, **next_state**)	if test_expr changes from start_state, then it can only change to next_state
n-Cycles	**assert_unchange**	#(severity_level, **width**, **num_cks**, **action_on_new_start**, property_type, msg, coverage_level)	(clk, reset_n, **start_event**, **test_expr**)	test_expr must not change within num_cks of start_event (action_on_new_start: 0=ignore, 1=restart, 2=error)
n-Cycles	**assert_width**	#(severity_level, **min_cks**, **max_cks**, property_type, msg, coverage_level)	(clk, reset_n, **test_expr**)	test_expr must hold for between min_cks and max_cks cycles
Event-bound	**assert_win_change**	#(severity_level, **width**, property_type, msg, coverage_level)	(clk, reset_n, **start_event**, **test_expr**, **end_event**)	test_expr must change between start_event and end_event
Event-bound	**assert_window**	#(severity_level, property_type, msg, coverage_level)	(clk, reset_n, **start_event**, **test_expr**, **end_event**)	test_expr must hold after the start_event and up to (and including) the end_event
Event-bound	**assert_win_unchange**	#(severity_level, **width**, property_type, msg, coverage_level)	(clk, reset_n, **start_event**, **test_expr**, **end_event**);	test_expr must not change between start_event and end_event
Single-Cycle	**assert_zero_one_hot**	#(severity_level, **width**, property_type, msg, coverage_level)	(clk, reset_n, **test_expr**)	test_expr must be one-hot or zero, i.e. at most one bit set high

PARAMETERS

severity_level
- `OVL_FATAL
- `OVL_ERROR
- `OVL_WARNING
- `OVL_INFO

property_type
- `OVL_ASSERT
- `OVL_ASSUME
- `OVL_IGNORE

msg descriptive string

USING OVL

+define+OVL_ASSERT_ON
+define+OVL_MAX_REPORT_ERROR=1
+define+OVL_INIT_MSG
+define+OVL_INIT_COUNT=<tbench>.ovl_init_count
+libext+.v+.vlib
-y <OVL_DIR>/std_ovl
+incdir+<OVL_DIR>/std_ovl

DESIGN ASSERTIONS

Monitors internal signals & Outputs

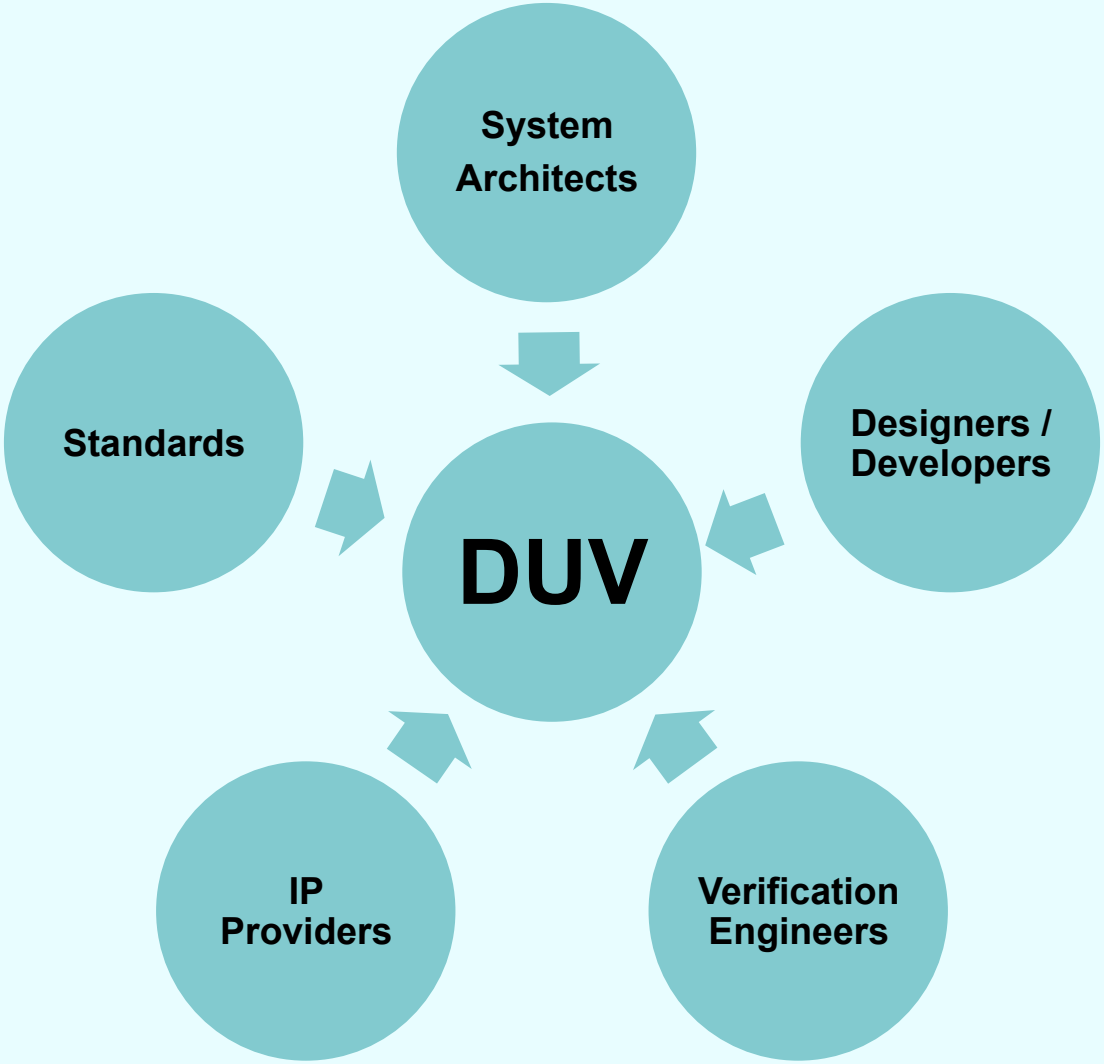
Examples
* One hot FSM
* Hit default case items
* FIFO / Stack
* Counters (overflow/increment)
* FSM transitions
* X checkers (assert_never_unknown)

INPUT ASSUMPTIONS

Restricts environment

Examples
* One hot inputs
* Range limits e.g. cache sizes
* Stability e.g. cache sizes
* No back-to-back reqs
* Handshaking sequences
* Bus protocol

Who writes the assertions?



Implementation Assertions

- **Also called "design" assertions.**
- **Specified by the designer/developer.**
  - Encode designer's assumptions.
    - Interface assertions
      - Catch different interpretations between different designers.
  - Formulate conditions of design misuse or design faults:
    - detect buffer over/under flow
    - signal read & write at the same time when only one is allowed
- Implementation assertions can detect discrepancies between design assumptions and implementation.

Specification Assertions

- **Also called "intent" assertions**
  - Often high-level properties.
- **Specified by architects, verification engineers, IP providers, standards.**
  - Encode expectations of the design based on understanding of functional intent.
  - Provide a "functional error detection" mechanism.
  - Supplement error detection performed by self-checking testbenches.
    - Instead of using (implementing) a monitor and checker, in some cases writing a block-level assertion can be much simpler.

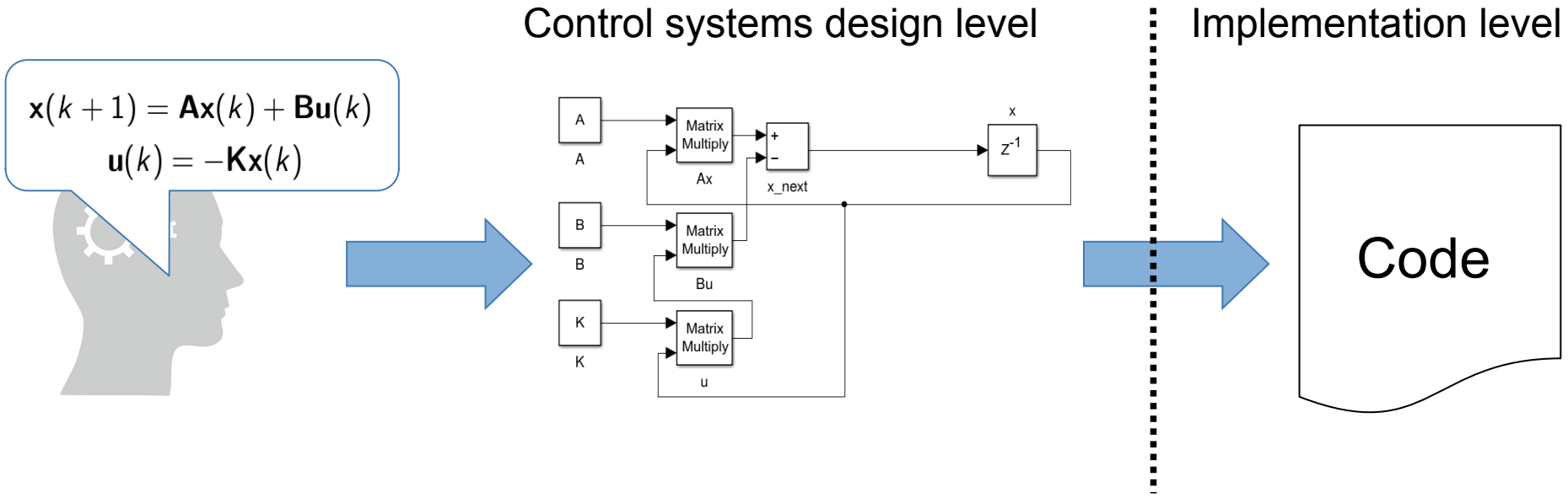
Do assertions really work?

- **Assertions are able to detect a significant percentage of design failures:** [Foster etal.: Assertion-Based Design. 2nd Edition, Kluwer, 2010.]

  - **34%** of all bugs were found by assertions on DEC Alpha 21164 project [Kantrowitz and Noack 1996]
  - **17%** of all bugs were found by assertions on Cyrix M3(p1) project [Krolnik 1998]
  - **25%** of all bugs were found by assertions on DEC Alpha 21264 project - The DEC 21264 Microprocessor [Taylor et al. 1998]
  - **25%** of all bugs were found by assertions on Cyrix M3(p2) project [Krolnik 1999]
  - **85%** of all bugs were found using OVL assertions on HP [Foster and Coelho 2001]

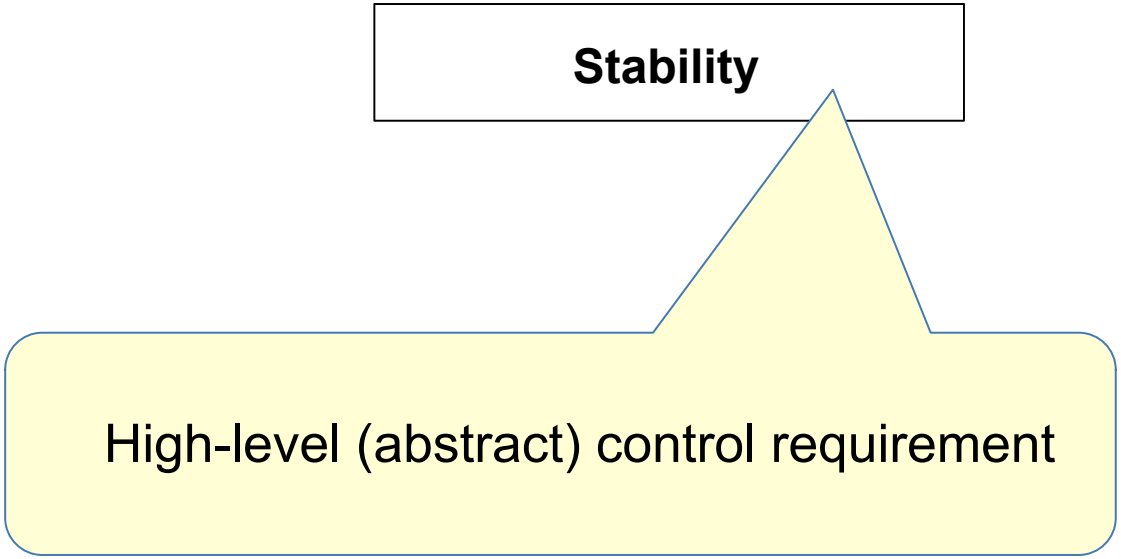
Assertions should be an integral part of a verification methodology.

Simulink Diagrams in Control Systems

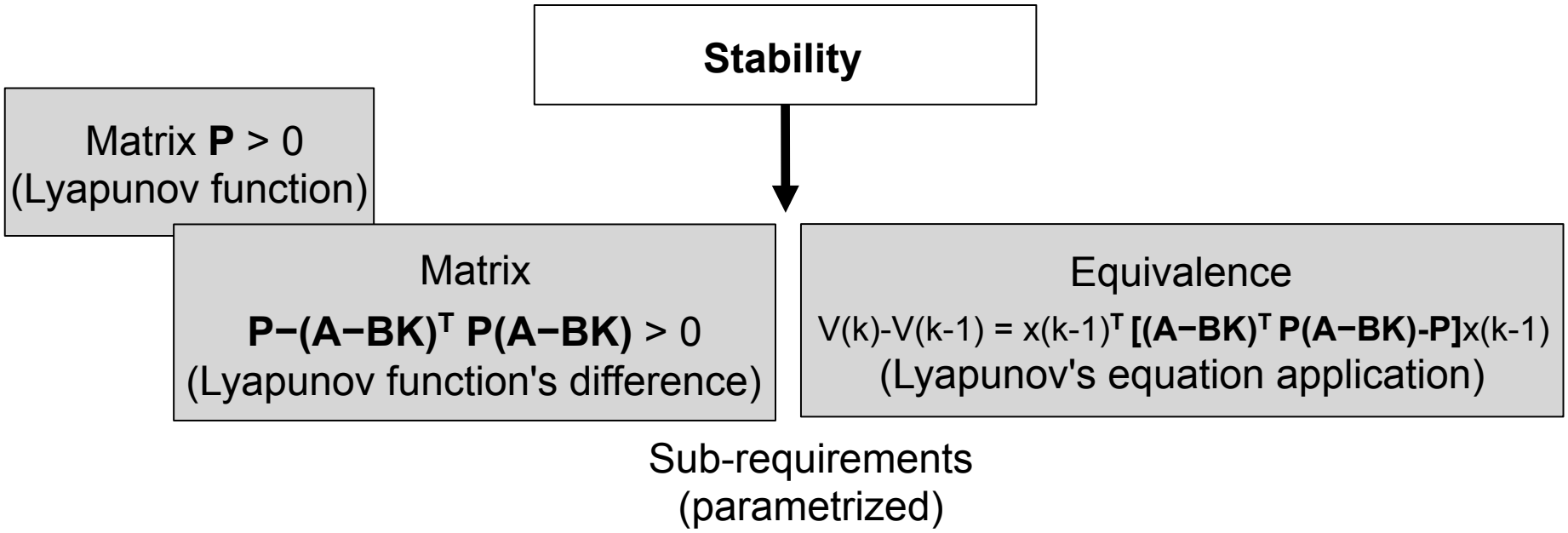


- Simulating the control systems
- Analysis techniques from control systems theory (e.g., stability)
- Serve as requirements/specification
- For (automatic) code generation

Verifying Stability



Verifying Stability



From control systems theory → Lyapunov's second method for stability:

Propose Lyapunov function that is
- Positive definite
- Monotonically decreasing

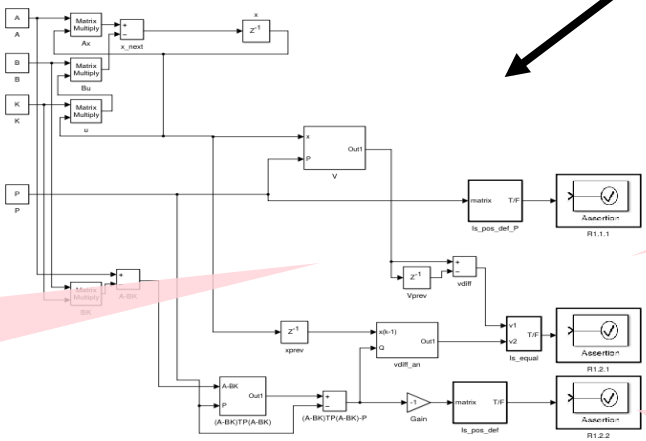
Verifying Stability

Stability

Matrix **P** > 0
(Lyapunov function)

Matrix
$\mathbf{P-(A-BK)^T\,P(A-BK)} > 0$
(Lyapunov function's difference)

Equivalence
$V(k)-V(k-1) = x(k-1)^T\,\mathbf{[(A-BK)^T\,P(A-BK)-P]}x(k-1)$
(Lyapunov's equation application)

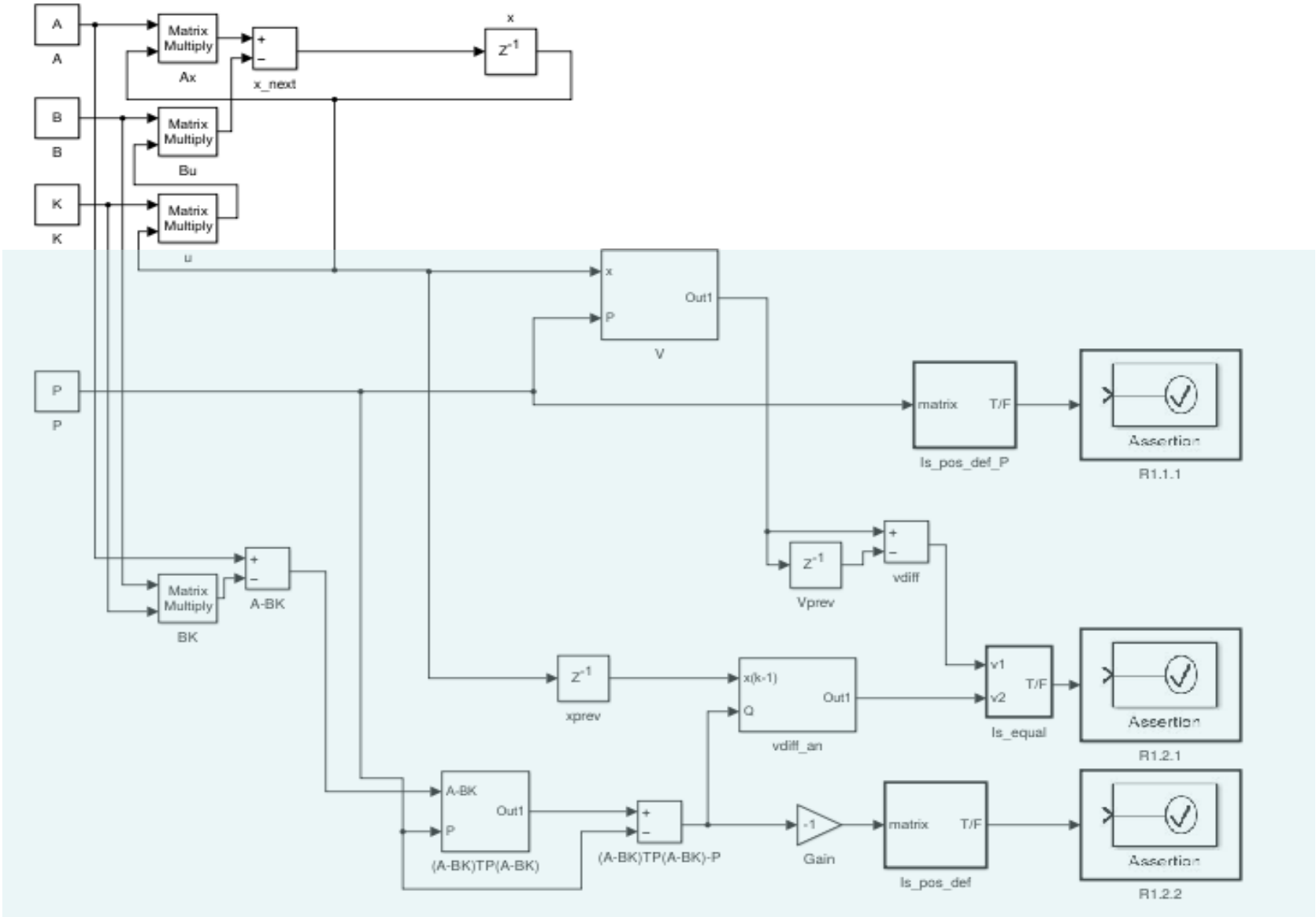


Capture control systems requirements

Add as assertions

Retain in code implementation

Assertion-Based Verification



Combining Verification Techniques

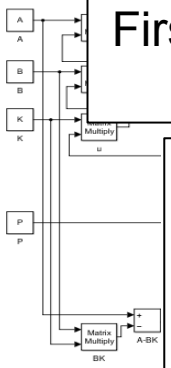
Stability

Matrix **P** > 0
(Lyapunov function)

Matrix
$\mathbf{P-(A-BK)^T P(A-BK)} > 0$
(Lyapunov function's difference)

Equivalence
$V(k)-V(k-1) = x(k-1)^T \mathbf{[(A-BK)^T P(A-BK)-P]} x(k-1)$
(Lyapunov's equation application)

First order logic theory of the Simulink diagram



Axiom: Bu = B * u
...

…

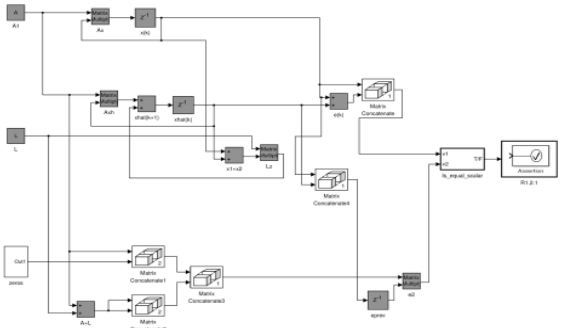
Goal: vdiff == vdiff_an

Test in simulation

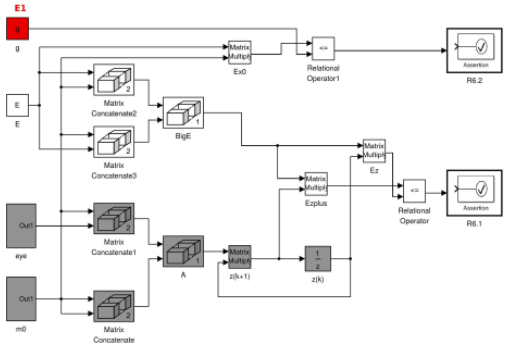
Automatic theorem proving

Case studies

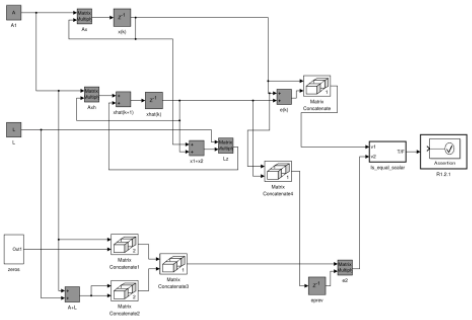
Estimators and controllers



Systems in series

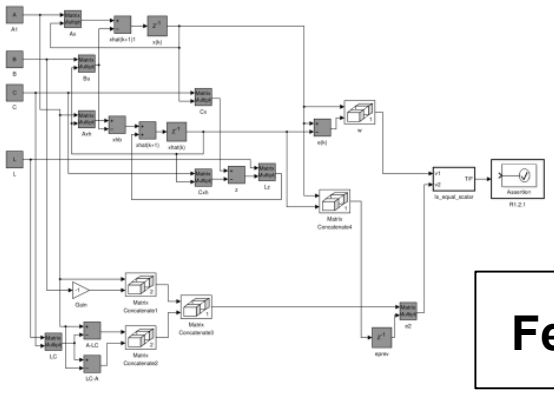


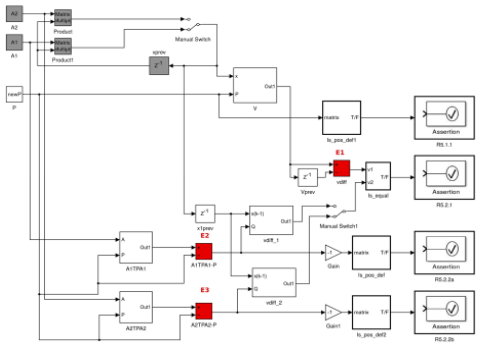
Functional equivalence



Stability

Hybrid systems



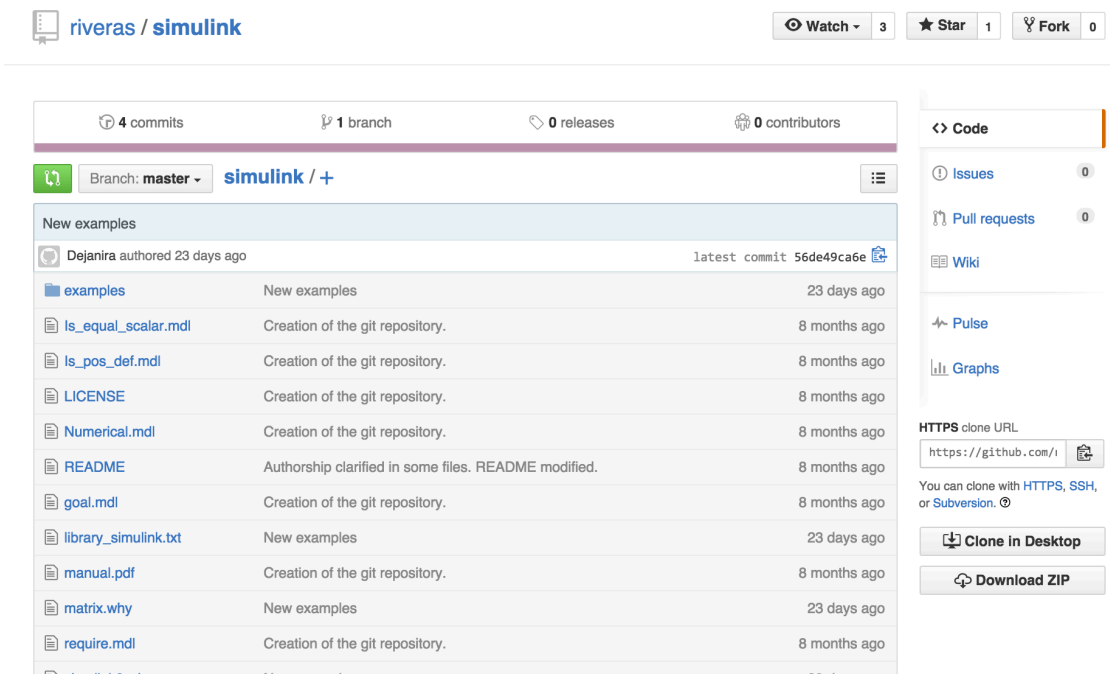


Feasibility (constraint satisfaction)

Moral

No single technique is adequate to cover a whole design in practice.

Combine techniques and learn from areas where verification is more mature.



http://github.com/riveras/simulink

D. Araiza Illan, K. Eder, A. Richards.
Formal Verification of Control Systems' Properties with Theorem Proving. International Conference on Control (CONTROL), pp. 244 – 249. IEEE, Jul 2014.
http://dx.doi.org/10.1109/CONTROL.2014.6915147

D. Araiza Illan, K. Eder, A. Richards.
Verification of Control Systems Implemented in Simulink with Assertion Checks and Theorem Proving: A Case Study.
European Control Conference (ECC), pp. tbc. Jul 2015.
http://arxiv.org/abs/1505.05699

DEMO 1:

How to verify control systems implemented in Simulink with assertion checks and theorem proving.

A simple case study.

Thank you

EPSRC
Engineering and Physical Sciences Research Council

Any questions?

Kerstin.Eder@bristol.ac.uk

Dejanira.AraizaIllan@bristol.ac.uk

Special thanks to Dejanira Araiza Illan, David Western, Arthur Richards, Jonathan Lawry, Trevor Martin, Piotr Trojanek, Yoav Hollander, Yaron Kashai, Mike Bartley, Tony Pipe and Chris Melhuish for their hard work, collaboration, inspiration and the many productive discussions we have had.

University of BRISTOL

brl
Bristol Robotics Laboratory