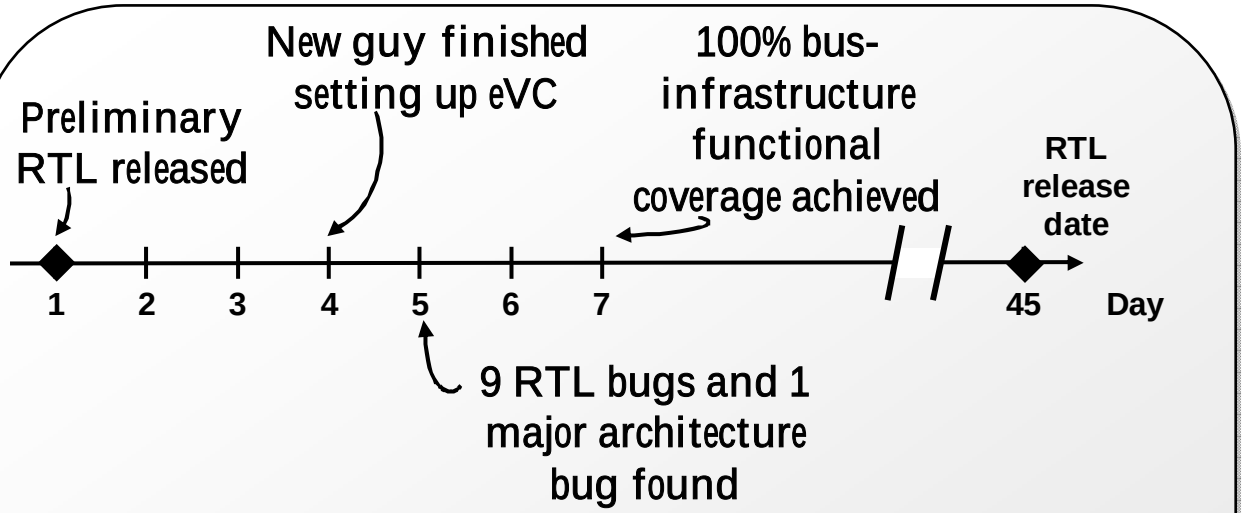
Simplifying SoC Verification using a Generic Approach



✓Reduce your verification effort by months

✓ Reduce your skill requirements

✓ Reuse proven solutions

✓ Make your verification quicker, easier, cheaper and safer

- SoC with 10 masters, 40 slaves, 7 buses, 3 bridges
- 4 days spent setting up testbench[1]
- 9 RTL bugs and 1 architecture bug found 1.5 months before the official RTL release date
- 7 days in total to get from RTL to 100% bus infrastructure functional coverage

[1] This was for a 1st time user - 2nd time users take about one day, depending on how fast they can type

david.robinson@verilab.com

What is it?

- An eVC that can be used to help verify any design
  - although of most benefit on bus based SoC designs
- It provides automatic bus verification and an advanced platform to use for the rest of the verification
- A database is used to hold information about the design, and APIs provide intelligent access to this information
- It contains labour saving features to let you focus on verification, not architecting and coding testbenches
- Best of class verification, testbench, coding and methodology knowledge is encapsulated and reusable by non-experts

- ✓ Get verifying immediately
- ✓ Keep verification off the critical path
- ✓ Deal effortlessly with design and specification changes
- ✓ Focus on verification - not on verification environments

Features

- Automatic bus verification
  - scoreboards, sequences, coverage, protocol limitations, automatic instantiation of bus agents, access limitations, safe and unsafe address accessing, register and memory map handling, bus traffic performance profiling, partially connected bus signals
- Productivity features
  - intelligent and filterable APIs to protect against design changes[1]
  - clock generation and verification, cold and warm reset generation, IRQ handling architecture, multi-dimensional virtual sequence architecture, easy linking of coverage to the VPlan[2], default generation of default slave address ranges, well defined configuration interface (including dynamic logging verbosity), end of test handling
  - provides a framework for incorporating and using other eVCs
  - excessive checking of user configuration, with error messages that suggest solutions
- Performance features
  - license and performance management through the working topology
  - code extensively optimised to reduce testbench overhead (esp. clock and reset generation)

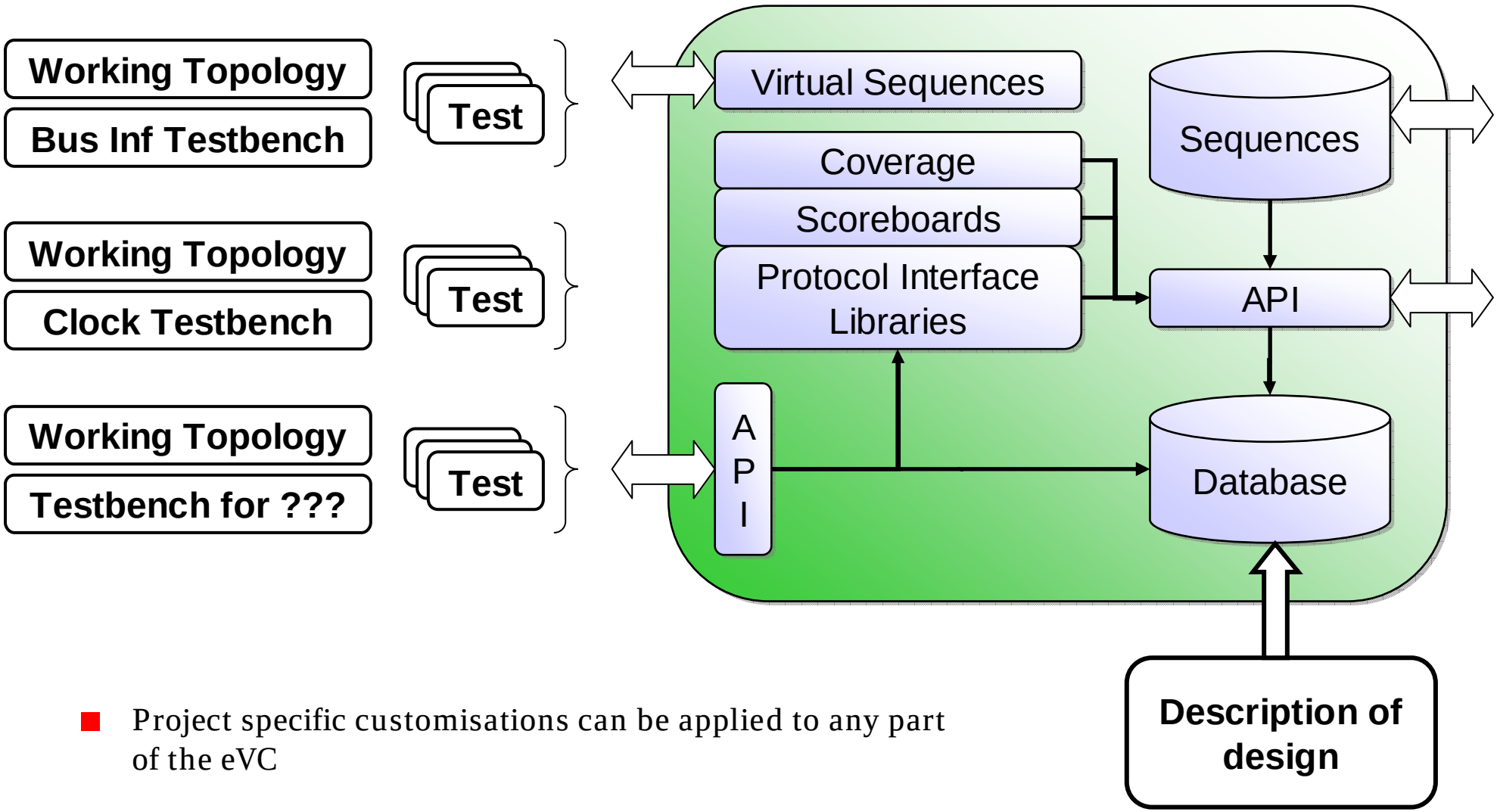
[1] e.g.:

```
.get_bus_interfaces(filter)
.get_address_ranges_for_slave(slave_name)
.get_all_masters_who_can_reach_me(slave_name,
                                  filter)
.get_bus_sequence_driver(name)
```

[2] e.g.: To create a reference from all active peripherals on AHB_BUS_1 that contain DMA in their name to the BASIC_TEST/DMA section of the VPlan:

```
filter.bus_names.add(AHB_BUS_1);
filter.mode = ACTIVE_ONLY;
coverage_file_writer.add_vplan_group_link(".*",
 ".*DMA.*", "BASIC_TEST/DMA", filter);
```

Basic Structure and Usage



- Project specific customisations can be applied to any part of the eVC

Describing the Design

- These are simply *e* "define as computed" macros

```
add master bus_interface{
 name     : CPU__INST_IF;
 component: CPU;
 bus      : INST_BUS;
 protocol : AHB;

 <limitations>;
  add direction: WRITE;
 </limitations>;

 <routing_table>;
  SRAM__S_IF;
  INST_BUS__DEFAULT_SLAVE_IF;
 </routing_table>;
};

ahb master INST_BUS CPU__INST_IF
        0 FALSE "hresetn" "hclk"
```

```
add bus{
 name         : INST_BUS;
 protocol     : AHB;
 multi-layer  : FALSE;
 address map  : MAIN;

 <limitations>;
   add size : TWO_WORDS;
   add size : FOUR_WORDS;
   add size : EIGHT_WORDS;
   add size : SIXTEEN_WORDS;
   add size : K_BITS;

   add slave_responses : RETRY;
   add slave_responses : SPLIT;
 </limitations>;
};

ahb bus INST_BUS "hresetn" "hclk";
```

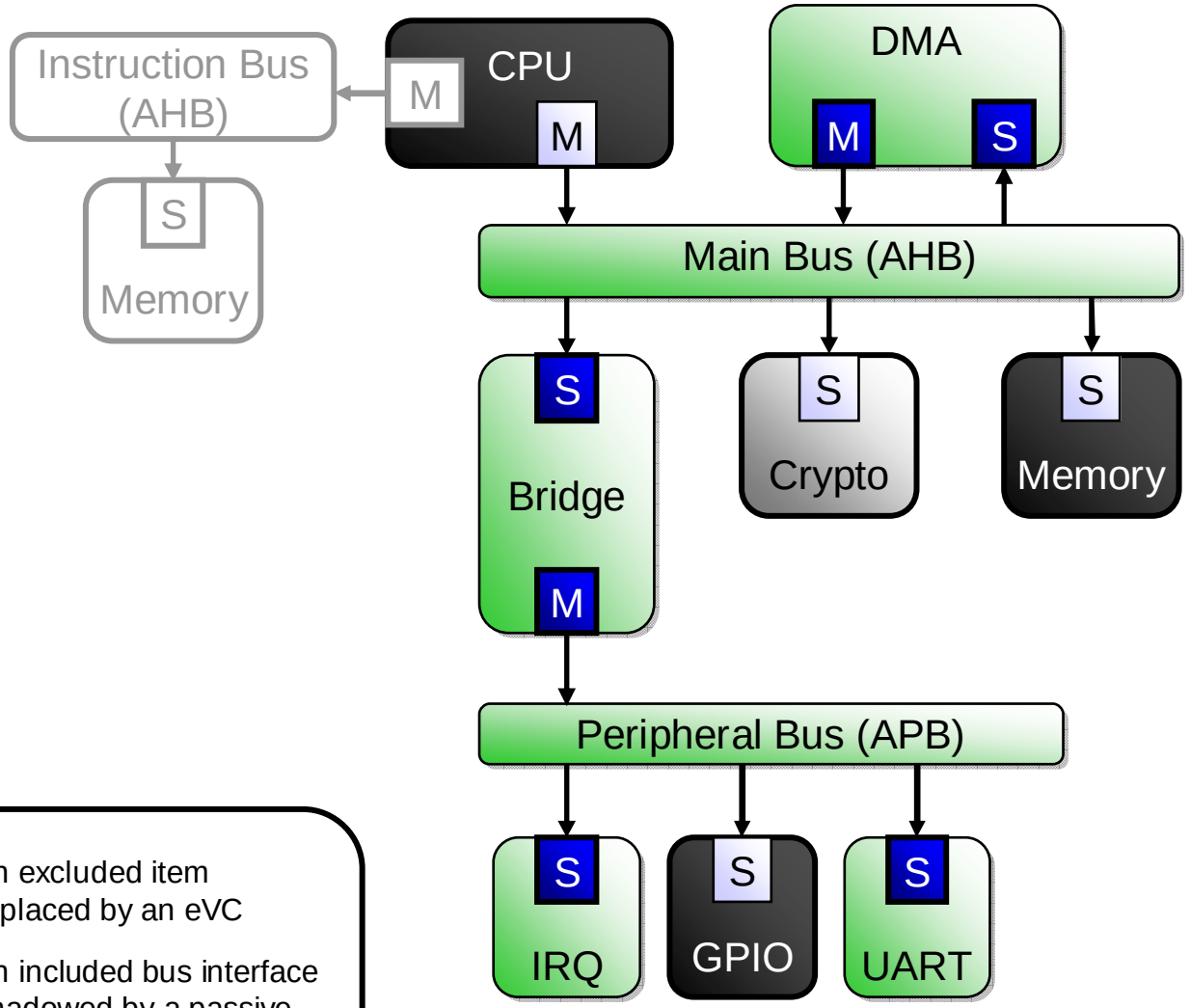
The Working Topology

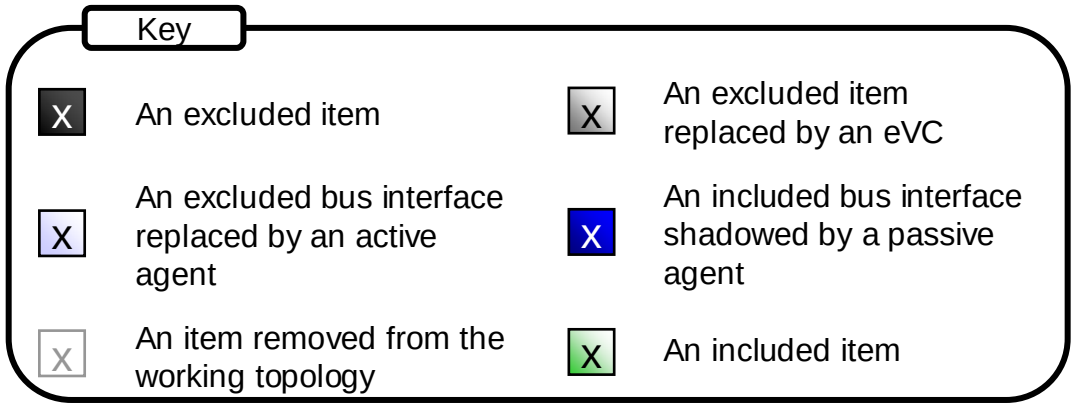
```
<use>
   AHB  t;  // use the AHB eVC
   AXI  f;  // don't use the AXI eVC
</use>

<subsystem>
   MAIN_BUS   t; // include
   INST_BUS   f; // exclude
   PERIPH_BUS t; // include
</subsystem>

<topology_item>
   bus hdl_included MAIN_BUS;
   bus hdl_included INST_BUS;
   com hdl_excluded CPU;
   bif hdl_excluded CPU__DATA_IF;
   bif hdl_excluded CPU__INST_IF;
</topology_item>
```

- Different testbenches need to see different views of the design
- Or alternatively: Different testbenches do not need to see the same view of the design
- The working topology file lets you specify:
  - which subsystems are to be included
  - which items are excluded from the HDL
  - which eVCs to use
- Possible Scenarios:
  - topology item removed from HDL and replaced with an eVC/Agent
  - topology item present in HDL and shadowed by an eVC/Agent
  - sub-systems removed entirely
  - eVCs excluded from the simulation (for license management and performance reasons)

Example Working Topology: Use Case Testing

1. The CPU (AHB agent) gets the DMA (RTL) to transfer data from the memory (AHB agent) to the Crypto (module eVC provides functionality)
2. The DMA then transfers this to the UART (RTL) ...
3. ... which issues interrupts ...
4. ....to the IRQ handler (RTL) ...
5. ... and these are picked up by the test using the in-built IRQ handling mechanisms





How do I do this Myself?

- Be aware of your language's limitations, and work out up-front how to work around them:
  - the unit structure of *e* makes it difficult to create dynamic topologies
  - SystemVerilog and SystemC lack:
    - the required macro features to create the required domain specific commands
    - the AOP features needed to create a truly flexible solution
    - a common register and address map methodology

- The following features are essential. The rest can be added at your leisure:
  - begin with the database, APIs and base scoreboards. Spend a lot of time getting these right, because changing them is a Big Deal. Build the filters into the APIs now. Test the API. Test them again
  - implement the protocol libraries for your buses. This involves creating the scoreboards, coverage generation, sequences and bus limitations. A protocol library can take a lot of effort to create but certain portions can be avoided if you don't mind losing flexibility
  - spend a lot of time on your error messages. Suggest solutions where possible. This eVC is aimed at non *e* programmers to use, so they won't be able to debug any configuration errors they make

- Avoid any hardwired information outside of the design description:
  - the more hardwired information there is, the less benefit you get from taking the generic approach