
Customer Methodology Flows for Model Checking:

Industry Experience and Practice with Affirma FormalCheck

The procedures described in this application note are broad and generic. Requirements for your specific design may dictate procedures slightly different from those described here.

Purpose

This application note describes:

- How two customers apply Affirma FormalCheck in their current processes
- How those customers select target designs to maximize the unique debugging value of Affirma FormalCheck
- Examples of what customers commonly prove with Affirma FormalCheck

FormalCheck Design Kit

Customer Methodology Flows for Model Checking:

Audience

The information presented here is intended to familiarize users with common usage of Affirma FormalCheck by major design houses, to help them develop their own usage model.

Overview

Affirma FormalCheck provides a unique capability for improving design quality, but because the technology is unfamiliar, customers sometimes are unsure how to best fit it into their methodology.

As a starting point, it is useful to mention the general guidelines we provide to customers, and then we will document the usage models of two customers that we have studied.

In general, we suggest that to obtain the most value from Affirma FormalCheck, it be used:

- As early as possible in the design process, by the designer, or by a verification specialist with good access to the designer. As FormalCheck requires no vectors, it can more easily be used in the early design process on a per-module basis. Many designers find that their design cannot only be debugged in FormalCheck, but also explored in interesting ways (for example, show me a way my design could get into some obscure state).
- As a complement to simulation; you can obtain greater effective coverage in a shorter time by taking the best of both technologies. To do this, you might select particular blocks for verification in FormalCheck which might

[Back](#)



Page 1-2



[Close](#)

FormalCheck Design Kit

Customer Methodology Flows for Model Checking:

be difficult to verify completely in simulation, such as arbiters/synchronizers, or blocks which must conform to a standard interface, especially if they will be re-used extensively, or if it would be damaging to your credibility/performance/market window if the standard is not met. With FormalCheck, you can in fact verify that in all cases, the spec compliance, or needed function, that you have described, is always satisfied, or you will be shown a failure waveform where it doesn't occur.

Below, we document two customers' usage models, and how they select blocks suitable for verification. They have now both been using FormalCheck continuously to improve the quality of their current designs for 1 year or longer.

One Usage Model

This customer employs FormalCheck in two primary modes, both on the pre-synthesis RTL netlist:

Mode 1)

In parallel with simulation during the initial design phase, currently being run by engineers who work closely with the designer. Customer has experience with various categories/sizes of designs, so they have gotten quite a good understanding which designs and properties are likely to be most valuable to formally verify.

Two valuable results:

1. Finding cases where design does not meet spec, and must be corrected.
2. Detecting errors/omissions in the specification.



FormalCheck Design Kit

Customer Methodology Flows for Model Checking:

This customer has found significant problems of both types, in approximately equal numbers, by using FormalCheck.

Mode 2)

In a regression mode, reusing previously created properties and constraints on a revised version of the design (such when design bug-fixes are made) to re-verify everything that was successfully verified on the prior version of the design. For this application, primarily the textual commands are used.

Value:

Properties/Constraints are largely reusable with little or no modifications, and allow any newly inserted bugs that interfere with performance of any previously verified behavior to be detected very efficiently. The textual interface provides an avenue for revising properties/constraints with a script, if the design has been changed extensively.

The customer most commonly creates new projects via the FormalCheck GUI, with the textual interface sometimes used in the property/constraint creation process, when this technique is preferable/efficient. They expect to have some users who work primarily just in the GUI, and some who work primarily via the FormalCheck textual interface.

This customer uses all of the basic FormalCheck features, and also clock extraction and the reduction manager.

Another Usage Model

A second customer has a strong commitment to defining a fully textual flow. This is used both for finding new bugs, and as regression test once completed. They

FormalCheck Design Kit

Customer Methodology Flows for Model Checking:

also verify the pre-synthesis RTL netlist, with some limited references to their cell-level library.

They have defined a complete process involving all of the FormalCheck textual commands, from initial project creation to verification, to analysis of the results. In addition, they have created example textual input files for FormalCheck that are used as templates, so the engineers can easily write files containing the behaviors to be proved and the assumptions that can be made.

The FormalCheck commands they use are attached as Appendix A.

There are actually multiple ways to use the textual commands; they are quite versatile. Another sample process is attached as Appendix B, to give an idea of an alternate approach.

They make some use of the FormalCheck GUI, sometimes for creating a property or constraint quickly; the graphical templates for capturing what is to be proven/assumed, and the design browser for selecting signals can be useful productivity aids. The Query Manager summarizes everything that needs to be proven about the design, whether it has been proven, not proven, is currently running, and if the result has become out of date and needs to be re-run.

Selection of Suitable Blocks for Verification

Customers develop experience which helps them select blocks; we also help advise them, and we have a paper “Verifying An Ethernet MAC” which discusses the topic in the context of a sample design project. We also mentioned some general guidelines earlier in this paper.

FormalCheck Design Kit

Customer Methodology Flows for Model Checking:

So what guidelines do customer use to decide which blocks to verify? Here is a sample from one customer:

Good candidates for verification:

- interacting finite state machines
- synchronizing blocks
- arbiters
- cache algorithms
- ECC codes
- mode selection state machines

Poor candidates for verification (simulated instead):

- performance registers
- simple fifos
- registers with straightforward read/write semantics.

Good candidates, from another customer:

- pipeline controller
- data transformation state machine
- bus interface units.

Some types of blocks (such as arbiters) are so commonly verified that we have appnotes on the topic. We also document a process, called “Query

FormalCheck Design Kit

Customer Methodology Flows for Model Checking:

Decomposition”, where a particularly large verification can be broken down into smaller parts, that can be proven separately.

Finally, it is worth mentioning that even if a block proves to be unsuitable for full verification for the reasons discussed, it is still possible to find some of the errors in the block this way, or to explore the normal operation of the block.

What to Prove?

Obviously, this depends on the type of device. The following examples are provided to give designers ideas about what they could verify. We will give them in an English-language form which is just like what you express in FormalCheck, whether via the GUI or textually:

For an arbiter:

1. Never more than one acknowledge is asserted at the same time.
2. There is no acknowledge without a request.
3. After a request there will eventually be an acknowledge.

For an ethernet media access controller:

1. All requests to transmit are acknowledged.
2. Never issue TransmitReady except after a TransmitRequest.
3. Flow control packets always have priority over data.
4. Normal data is never transmitted when the transmitter is paused.

FormalCheck Design Kit

Customer Methodology Flows for Model Checking:

5. The minimum interframe time between flow control packets is large enough to allow normal data packets to be transmitted.
6. Never issue TransmitReady unless the next state is normal data transmission.

For more detail, see our Ethernet MAC white paper. Note that ideas for queries were made by relying heavily on the Ethernet spec, and thinking about correct operation, and critical aspects of spec compliance, such as performance, and not losing data.

General common properties you might ask about any device:

1. Always $S_0, S_1, S_2,$ or S_3 should be 1.
2. Never more than one of S_0, S_1, S_2, S_3 should be 1.
3. Whenever $W=1,$ $A(8 \text{ downto } 0)$ should be stable.
4. Always when $X=1,$ $Y(15 \text{ downto } 0)=0xDEAD$
5. After $C=1,$ eventually $A=1.$
6. After $C=1$ and $D=1,$ $A=1$ within n clock cycles.
7. After every transition on $A(8 \text{ downto } 0),$ $D=1$ for one clock cycle.

Remember that by verifying properties that describe the correct function of your device, you could find bugs in any part of the design that can ever malfunction, that could cause that property to fail. So you really don't have to write properties describing every possible illegal condition of your device.

One really useful method is to write down the 10 or 20 most important functions your block has to provide, for a medium-complexity block, and then schedule all

FormalCheck Design Kit

Customer Methodology Flows for Model Checking:

of these using the new “Check!” algorithm, which will quickly find any easily found bugs in the design. You can focus on debugging these from the error waveforms, and get a lot of value from your verification efforts right away.

When the easy bugs have been fixed, it is easy to switch over to “Verify!” to find the difficult bugs, or verify that the needed behavior holds in all cases.

Conclusion

We have described common FormalCheck usage at two customer sites, their block selection criteria, and provided ideas about what to prove within FormalCheck. The objective of this is to give new customers ideas how to begin with FormalCheck, since it does represent a new and somewhat unfamiliar technology, but also a very powerful one that is proving valuable in production design work on a daily basis.

Appendix A: One Customer’s Textual FormalCheck Flow

Not all steps are needed; this just provides their designers an idea of the options and flexibility that are available. Some of these steps are written into a makefile that is used by an automatic regression tests of any revised designs, to catch newly inserted bugs as they occur.

Initially specifying the design files, design root, target language, and
possibly some initial assumptions about legal inputs to the design.

```
fcimport -p proj.fpj proj.text
```



FormalCheck Design Kit

Customer Methodology Flows for Model Checking:

Compile the design model

```
fcbuild -p proj.fpj
```

You may export a pre-existing query from the project file for editing.

```
fcexportquery -p proj.fpj sample_query_name
```

You may edit your query file sample_query_name.qry and verify it by:

```
fcverify -p proj.fpj -r sample_query_name  
sample_query_name.qry
```

Generate a report the for the query

```
fcreport -p proj.fpj -o sample_query_name.report -a  
sample_query_name
```

(or report for whole project by omitting “sample_query_name”)

Generate design coverage data for the query

```
fcdcov -p proj.fpj sample_query_name
```

(or for all queries by omitting “sample_query_name”)

Generate the VCD file

```
fcview -VCD -p proj.fpj sample_query_name
```

Run signalscan waveform viewer

```
signalscan sample_query_name.vcd
```

Appendix B: Another Sample FormalCheck Textual Flow

1. Compile (build) the HDL design; create the .fpj project file

```
fcbuild -r <root> -L <lang> -p <proj_file>  
-I <include> <files>
```

2. Create ASCII text file(s) containing macros, state vars, constraints in ASCII project file format.

3. Import the ascii file(s) created in step 2.

```
fcimport -p <proj_file> ascii_file
```

If you want to keep them all in separate files, you can import each one separately, or create a temp file that sources the other files, and import that one, like this:

e.g. file “ascii_file”:

```
source macros.asc
```

```
source constr.asc
```

```
source stvars.asc
```

4. Create ASCII text file(s) containing queries in PROJECT file format (an example is shown in Appendix C)

5. Import the ascii query file(s)

FormalCheck Design Kit

Customer Methodology Flows for Model Checking:

```
fcimport -p <proj_file> <query_file1>
```

```
fcimport -p <proj_file> <query_file2>
```

6. Verify the query(ies) just imported

```
fcverify -r <query_name_list> -p <proj_file>
```

where <query_name_list> is query1,query2, query3..., or omit -r to verify all queries

7. Generate a project report to see the results of the verification

```
fcreport -a -p <proj_file>
```

8. If query failed-- bring up the waveform viewer to start debug process

```
fcview -o -p <proj_file> <name_of_failed_query>
```

9. Re-build the design after changes are made (i.e. to fix the bug)

```
fcbuild -p <proj_file.fpj>
```

10. Go to step 6 to reverify query(ies)

11. When query is verified, examine coverage for the query:

```
fcdcov -p <proj_file> <name_of_verified_query>
```

Appendix C: Simple Example of a Textual Query

The query itself, and the names of all the needed constraints, in the format supported by “fcimport” (additional detail can be found in “The Affirma FormalCheck User’s Guide, Appendix D”):

```
Query Read_Hit{  
  
    Ensure Read_Hit {  
  
        After {@Read && @Ads && @ClockRising && @Match}  
  
        Always {ci.cfs == 1}  
  
        Unless {ci.cfs == 0}  
  
    }  
  
    Constraints {  
  
        Valid_Data Seq_for_Ads Stable_bus_addr Stable_op  
        Valid_Cpu_Op Valid_Cache_Op clk reset  
  
    }  
}
```