Utilizing Vera Functional Coverage in the Verification of a Protocol Engine for the FlexRay™ Automotive Communication System

Mark Litterick, Verilab
&
Markus Brenner, Freescale Semiconductor

mark.litterick@verilab.com
markus.brenner@freescale.com

ABSTRACT

The FlexRay™ communication system for advanced automotive control applications such as drive-by-wire is specified at a micro-architectural level using graphical Specification and Description Language (SDL) representations for the main protocol engine operation. Verifying that a specific implementation conforms to this specification requires close attention to all possible state transitions and conditional paths, for a potentially huge variety of node and cluster configurations. This paper explores the issues involved with measuring and managing the functional coverage for a FlexRay™ controller using Vera. Practical solutions and code examples are presented based on the verification environment for a FlexRay™ protocol engine implementation at Freescale Semiconductor.

Table of Contents

Table of Figures

1 Introduction

The FlexRay[1] communication system is an emerging standard for advanced automotive control applications, such as drive-by-wire. The detailed micro-architectural level of the FlexRay specification facilitates implementations where the internal operation closely reflects the protocol specification definitions. Corresponding functional coverage models in the verification environment can also be defined relative to the FlexRay specification which enables consistent understanding, easier maintenance and better reuse. This paper explores the general issues of functional coverage pertaining to the FlexRay specification and then presents a solution using Vera.

The paper is structured as follows. Section 2 provides an overview of the FlexRay protocol with a particular focus on the characteristics of the protocol specification which are later exploited in the coverage model. In Section 3 the generic role of functional coverage in relation to other aspects of a verification environment is discussed and appropriate application of different types of functional coverage to aspects of FlexRay protocol operation are suggested. Section 4 takes a close look at defining coverage points for one of the key components of the protocol engine. The architectural implementation of Vera coverage classes and related verification components is provided in Section 5 with some example code. Section 6 provides an overview on managing the coverage data in the presence of diverse FlexRay cluster configurations and in Section 7 the paper is concluded.

2 Overview of FlexRay Protocol

The key aspects of the FlexRay communications system include flexibility, performance and fault tolerance. The FlexRay Consortium homepage [1] provides considerable background information on the communication system goals, capability and status. This section of the paper provides a brief overview of FlexRay operation and specification as it relates to the functional coverage task. Readers interested in the full details of FlexRay operation can request a copy of the FlexRay Protocol Specification [2] from the Consortium homepage.

FlexRay systems support a number of cluster topologies with different nodes connected to several independent physical channels. Communication between the nodes in the cluster is achieved by encoding the payload data into frames with associated header information and error checking codes. These frames are serially transported as part of a higher-level communication cycle containing a static Time Division Multiple Access (TDMA) segment and an optional Flexible TDMA dynamic segment. A startup protocol establishes the communication cycle timing between several nodes in the system and allows for other nodes to integrate to this schedule. Each node then monitors and adjusts its clock rate and offset continuously to achieve acceptable clock synchronization for the duration of all subsequent communication cycles.

A key characteristic of the FlexRay protocol specification is the micro-architectural definition of protocol operation using graphical Specification and Description Language (SDL) (refer to [3] for useful information on SDL). This detailed level of specification facilitates implementations which make use of protocol constants, parameters and variables resulting in operation that is traceable to the specification SDL.

[1] The word FlexRay is a registered trademark of the FlexRay Consortium.

The Protocol Operation Control (POC) mechanism is central to the overall functionality and is responsible for synchronizing and coordinating all protocol behavior by interfacing with the system Controller Host Interface (CHI) and the core mechanisms of coding/decoding, media access control, frame/symbol processing, clock synchronization and bus guardian schedule monitoring. The state transition diagram shown in Figure 1 provides an overview of the POC operation.

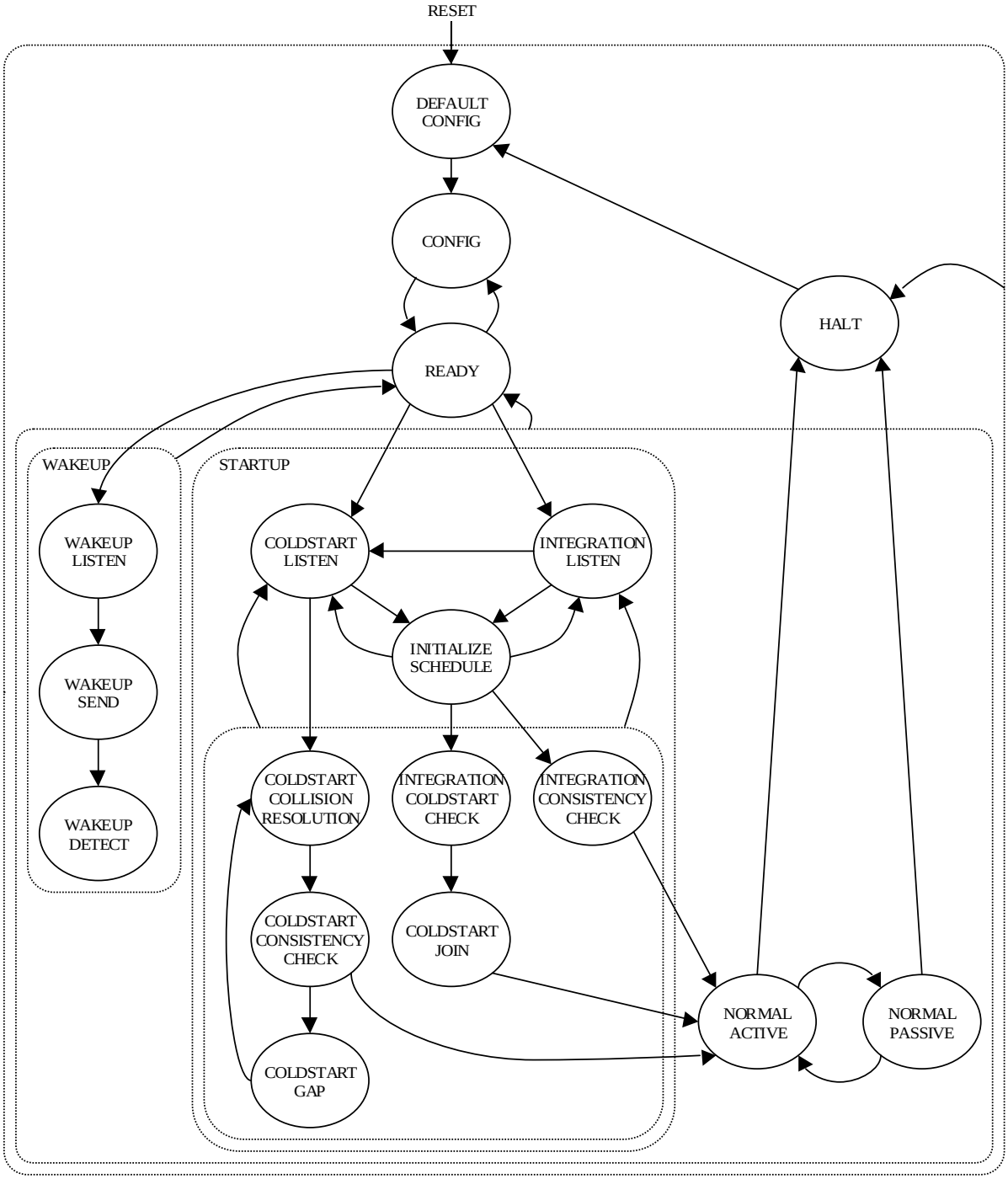

Figure 1: Overview of POC States and Transitions

The real complexity of POC operation is however not immediately apparent from Figure 1. For instance in each POC state there are a large number of potential trigger events that can

cause transitions to other states, or modify protocol behavior in some other way. Figure 2 depicts an example SDL diagram, which has been taken from *Figure 7-11* of the FlexRay Protocol Specification and annotated with tags (depicted as callout boxes with reference numbers, e.g. F07-11-003) to clearly reference each step in the SDL. In this case there are many possible trigger events (depicted with the ⟨ symbol) that are relevant when the POC is in the *coldstart_listen* state.

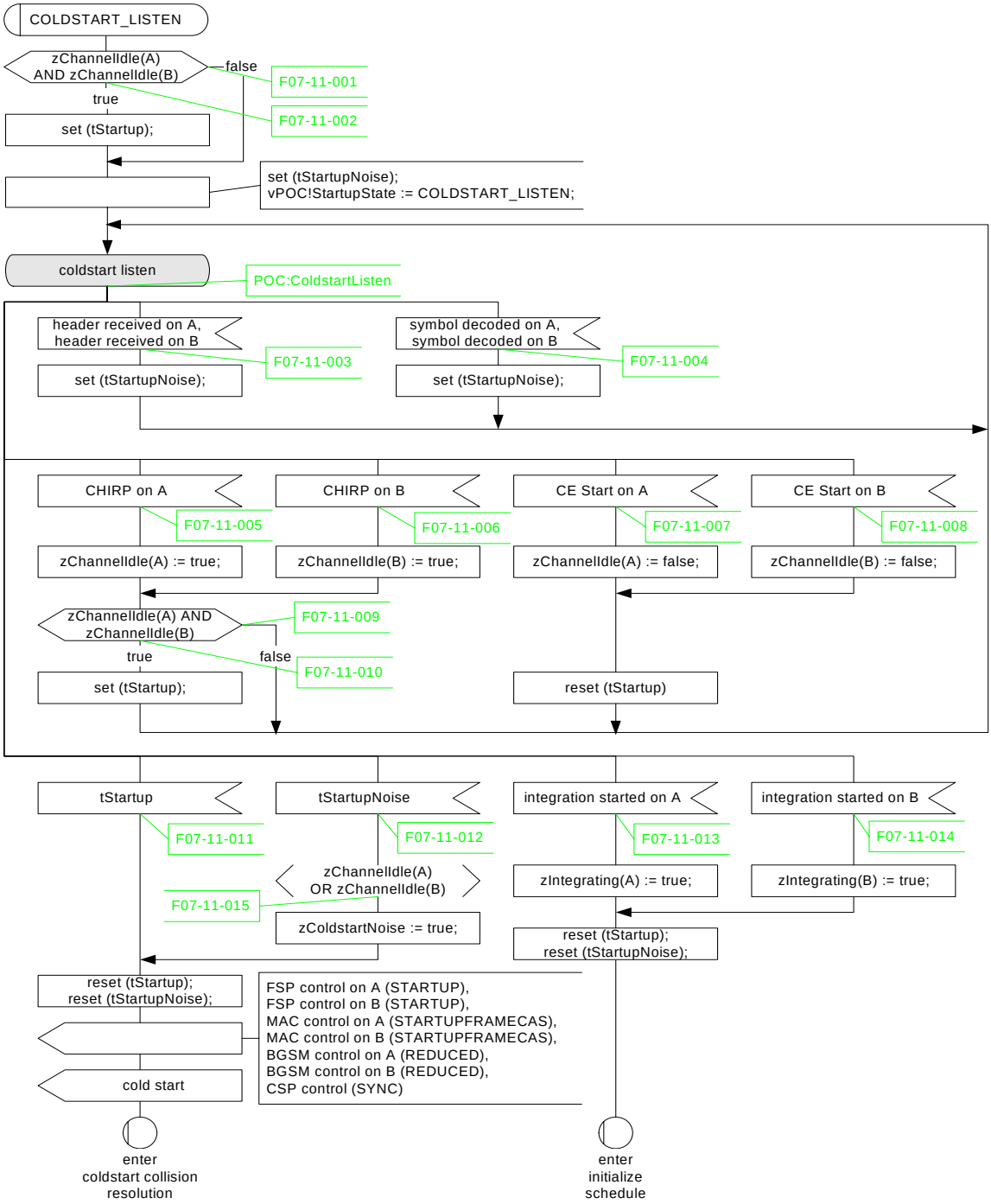

Figure 2: SDL for POC *coldstart_listen* State

Likewise there can be multiple potential paths for the transitions between POC states depending on different conditions (depicted by the ⟨⟩ symbol), an example of which is

shown in Figure 3, which is taken from *Figure 7-13* of the FlexRay Protocol Specification and annotated with tags. Note that *abort startup* in this diagram refers to an SDL macro which is used to determine if the next state is *coldstart_listen* or *integration_listen*. In Figure 3 there are three potential paths between the *coldstart_consistency_check* state and *integration_listen* (via *abort startup*) depending on the current values of the *vCycleCounter*, *zStartupNodes*, *vRemainingColdstartAttempts* and *zSyncCalcResult* variables, for example via tags *F07-13-007*, *F07-13-009* or *F07-13-011*.

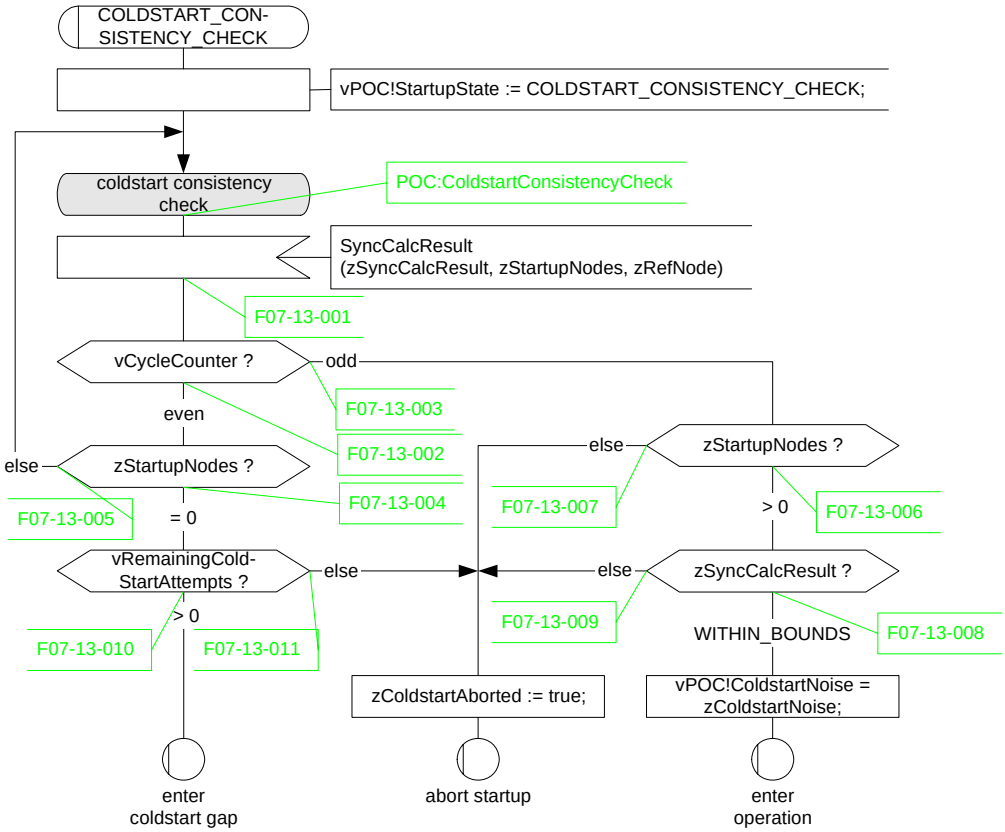

Figure 3: SDL for POC *coldstart_consistency_check* state

There are about 24 SDL diagrams in the protocol specification used to specify the behavioral operation of the POC alone, and many more covering the functionality of the other core mechanisms. The only major functionality of the FlexRay Protocol Specification that is not covered by SDL diagrams is the frame format definition and CHI operation.

3 The Role of Functional Coverage

Coverage is the mechanism by which the effectiveness of a verification environment is measured. In an effective Vera verification environment functional coverage constitutes an essential part of the methodology but the complimentary techniques of code coverage and functional checking are also required.

Code coverage is used to determine how thoroughly the RTL code was exercised by the test suite, in terms of statements, branches and conditions executed. Code coverage is necessary but not sufficient; it does not tell you if the stimulus was functionally representative, or whether the DUT responded correctly to the stimulus, and most importantly code coverage tells you nothing about functionality that has been accidentally omitted (i.e. missing code).

One benefit of code coverage is that it does not involve manual specification of coverage points; it examines every aspect of RTL code within the scope of its capabilities. Typically code coverage is applied late in the design cycle when the regression test suite and RTL are both mature.

Functional checking is used to determine if the DUT responded correctly to the test stimulus. Even if 100% functional coverage is achieved over an exhaustive set of coverage points, it is still necessary to validate the functional correctness of the DUT operation using checkers.

Functional coverage is used to determine the thoroughness with which interesting and relevant scenarios have been applied to, and processed by, the DUT. The criteria for functional coverage are extracted manually from the specifications and verification test plan. Functional coverage can be applied from an early stage in the verification cycle and used to guide (automatically or manually) the generation of interesting test conditions.

In agreement with the guidelines provided by the Synopsys Reference Verification Methodology (RVM) [4] a complete functional coverage model includes both stimulus and response aspects, for example:

- stimulus to the DUT
- internal state of the DUT
- response from the DUT

Stimulus coverage provides a measure of how effectively the verification environment generated all relevant and interesting scenarios and corner cases, but does not determine whether the DUT responded correctly to the stimulus. State coverage is concerned with major protocol steps and sequences; it is a measure of whether the verification environment was able to drive the DUT through all protocol corner cases and relevant state space. Response coverage is really a mechanism for validating completeness of the checking infrastructure; for example coverage is used to determine whether all possible output responses were generated under different conditions, whereas checkers are used to validate correct behavior of generated outputs.

This paper focuses on the implementation and overall contribution of the internal state functional coverage for the Protocol Operation Control (POC) for a FlexRay protocol engine since it plays such a significant part in the micro-architectural definition of the protocol specification. State coverage is also appropriate for the other core mechanisms in the protocol which are specified using SDL, for example clock synchronization. The frame format and CHI aspects of the specification are most appropriately handled by stimulus and response coverage.

4 POC Coverage

An analysis of the relevant SDL diagrams in the FlexRay specification implies that two aspects of state coverage have to be considered in order to fully cover the POC operation:

- trigger events which cause the POC to change state
- prevalent conditions when the POC changes state

The main reason that these requirements are considered separately for coverage is related to timing. Trigger events in the SDL result in subsequent state changes; however when the actual state change occurs in a real RTL implementation the trigger event may no longer be active. The system variables and configuration parameters are stable when the state change occurs and can be used to determine which path the POC took between the two states.

4.1 POC Trigger Coverage

In order to measure the trigger coverage it is necessary to determine if all relevant SDL trigger events occurred during each appropriate state. This can be achieved in Vera by defining a conditional *state* bin. Full details of coverage bin definitions are given in the OpenVera Language Reference Manual [5]; a summary of *state bin_type* definition syntax is shown in Figure 4.

```
[wildcard] state bin_name (specification) [conditional];
```

Figure 4: Vera State Bin Definition

There are approximately 220 such trigger coverage definitions required to capture all of the relevant POC SDL definitions, each of which is identified with a *bin_name* which corresponds to an appropriate specification tag. Figure 5 shows the trigger coverage definitions required for the POC *coldstart_listen* state, the SDL for which is shown in Figure 2. The trigger coverage *sample_event* is activated whenever any of the POC trigger events are active.

```
#define POC_CL 7'b111_0011
sample poc.vState {
  ...
  state F07_11_003_A (POC_CL) if (poc.header_received_on_A === 1);
  state F07_11_003_B (POC_CL) if (poc.header_received_on_B === 1);
  state F07_11_004_A (POC_CL) if (poc.symbol_decoded_on_A === 1);
  state F07_11_004_B (POC_CL) if (poc.symbol_decoded_on_B === 1);
  state F07_11_005   (POC_CL) if (poc.CHIRP_on_A === 1);
  state F07_11_006   (POC_CL) if (poc.CHIRP_on_B === 1);
  state F07_11_007   (POC_CL) if (poc.CE_Start_on_A === 1);
  state F07_11_008   (POC_CL) if (poc.CE_Start_on_B === 1);
  state F07_11_011   (POC_CL) if (poc.tStartup === 1);
  state F07_11_012   (POC_CL) if (poc.tStartupNoise === 1);
  state F07_11_013   (POC_CL) if (poc.integration_started_on_A === 1);
  state F07_11_014   (POC_CL) if (poc.integration_started_on_B === 1);
  ...
```

Figure 5: POC Trigger Coverage Definitions for *coldstart_listen* State

Note that case equality (===) should be used in the conditional expressions since 'x' or 'z' values in the member variables of the *poc* object would result the condition evaluating to an undefined value if logical equality (==) were used. Such undefined conditions lead to erroneous coverage results with false hits being reported.

4.2 POC State Coverage

The primary concern for POC state coverage is determining if all the states have been reached and if all conditional paths between the different states have been exercised. To achieve this, state coverage is broken down into four groups of bin definitions:

- states
- state transitions
- conditional state transitions
- state transition sequences

The basic state bin definitions are defined using wildcard *state bin_type* definitions as shown in Figure 6. The *bad_state bin_type* definition shown provides a checking function by generating a Vera error if the sampled value for *poc.vState* does not match one of the specified state definitions in the *coverage_group*.

```
sample poc.vState {

  wildcard state DEFAULT_CONFIG                 (7'b000_xxxx);
  wildcard state CONFIG                         (7'b001_xxxx);
  wildcard state READY                          (7'b011_xxxx);
  wildcard state WAKEUP_LISTEN                  (7'b010_xx01);
  wildcard state WAKEUP_SEND                    (7'b010_xx11);
  wildcard state WAKEUP_DETECT                  (7'b010_xx10);
  wildcard state COLDSTART_LISTEN               (7'b111_0011);
  wildcard state COLDSTART_COLLISION_RESOLUTION (7'b111_0010);
  wildcard state COLDSTART_CONSISTENCY_CHECK    (7'b111_1010);
  wildcard state COLDSTART_GAP                  (7'b111_1110);
  wildcard state INITIALIZE_SCHEDULE            (7'b111_0111);
  wildcard state INTEGRATION_COLDSTART_CHECK    (7'b111_1101);
  wildcard state COLDSTART_JOIN                 (7'b111_1111);
  wildcard state INTEGRATION_LISTEN             (7'b111_0101);
  wildcard state INTEGRATION_CONSISTENCY_CHECK  (7'b111_0100);
  wildcard state NORMAL_ACTIVE                  (7'b101_xxxx);
  wildcard state NORMAL_PASSIVE                 (7'b100_xxxx);
  wildcard state HALT                           (7'b110_xxxx);

       bad_state ILLEGAL_STATE                  (not state);
```

Figure 6: POC State Bin Definition

In Vera *trans bin_type* definitions can be used to specify coverage points for state transitions; a summary of the syntax is given in Figure 7. In order to measure such state transitions it is necessary to activate the corresponding *sample_event* only when a POC state change occurs.

```
trans bin_name (“state_1” -> “state_2” [-> “state_N”]) [conditional];
```

Figure 7: Vera Transition Bin Definition

All possible basic state transitions are specified (without conditions) to allow for the *bad_trans bin_type* to be used to check for illegal transitions between states. The POC definition requires about 90 such legal state transitions to be defined. Figure 8 shows an example of some basic state transition definitions.

```
trans DC_C    ("DEFAULT_CONFIG" -> "CONFIG");
trans C_R     ("CONFIG"         -> "READY");
trans R_C     ("READY"          -> "CONFIG");
trans R_WL    ("READY"          -> "WAKEUP_LISTEN");
...
bad_trans ILLEGAL_TRANS (not trans);
```

Figure 8: POC State Transition Bin Definition

The main set of conditional state transition bin definitions for the POC SDL comprises about 170 coverage points. Figure 9 shows the conditional state transition coverage definitions required for transitions between the POC *coldstart_consistency_check* and *integration_listen* states, the SDL for which is shown in Figure 3.

```
trans F07_13_007_IL ("COLDSTART_CONSISTENCY_CHECK" -> "INTEGRATION_LISTEN")
                     if ((poc.zStartupNodes <= 0)
                      && (poc.vCycleCounter[0] === 1)); // F07_13_003

trans F07_13_009_IL ("COLDSTART_CONSISTENCY_CHECK" -> "INTEGRATION_LISTEN")
                     if ((poc.zSyncCalcResult !== WITHIN_BOUNDS)
                      && (poc.zStartupNodes > 0)        // F07_13_006
                      && (poc.vCycleCounter[0] === 1)); // F07_13_003

trans F07_13_011_IL ("COLDSTART_CONSISTENCY_CHECK" -> "INTEGRATION_LISTEN")
                     if ((poc.vRemainingColdstartAttempts <= 0)
                      && (poc.zStartupNodes === 0)      // F07_13_004
                      && (poc.vCycleCounter[0] === 0)); // F07_13_002
```

Figure 9: Conditional Transitions from *coldstart_consistency_check* to *integration_listen*

In addition to the basic and conditional state transitions it is also possible to specify state transition sequences in Vera. This is useful in our application to provide a check that the POC actually performed the correct sequence of state changes at different stages in protocol operation; if required the testcase can check if the sequence was successful by calling a *queryBin()* method as described in Section 5 . About 20 or so state transition sequences are required for the general POC functionality an example of which is shown in Figure 10.

```
trans POC_CSA_CSI     (   "READY"
                       -> "COLDSTART_LISTEN"
                       -> "INITIALIZE_SCHEDULE"
                       -> "INTEGRATION_COLDSTART_CHECK"
                       -> "COLDSTART_JOIN"
                       -> "NORMAL_ACTIVE");
```

Figure 10: POC State Transition Sequence Bin Definition

5 Vera Implementation

The Vera implementation is based around a class hierarchy which makes use of existing base classes for data-objects and monitors as shown in the UML class diagram of Figure 11 (refer to [6] for a useful quick reference to UML).

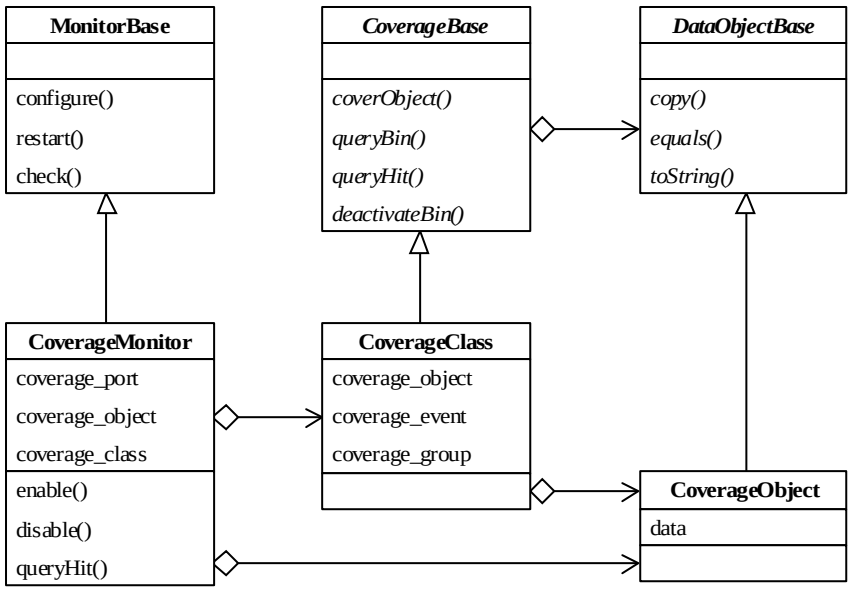

Figure 11: Coverage Class Hierarchy

CoverageObjects are used to transport information from the CoverageMonitors to the *coverage_groups* which are contained in the CoverageClasses. The CoverageObjects are all extended from the DataObjectBase class which allows the CoverageBase class to be defined as a virtual class as shown in Figure 12. This approach provides a standard generic pattern for measuring coverage that can be appropriately specialized in each of the derived subclasses.

```
virtual class CoverageBase {
  virtual task coverObject(DataObjectBase obj);
  virtual function integer queryBin(string bin_pattern);
  virtual function integer queryHit(string bin_pattern);
  virtual task deactivateBin(string bin_pattern);
}
```

Figure 12: CoverageBase Class Definition

The CoverageObjects are used as containers for all the information to be covered. Figure 13 shows an excerpt from the FRPocState CoverageObject definition:

```
class FRPocState extends DataObjectBase {
  // POC state
  public bit[6:0]   vState;
  ...
  // cluster and node configuration parameters (~8 definitions)
  public channelIdT pWakeupChannel;
  ...
  // protocol variables (~7 definitions)
  public integer vRemainingColdStartAttempts;
  ...
  // process variables (~10 definitions)
  public syncCalcResultT zSyncCalcResult;
  ...
  // Trigger signals (~36 definitions)
  public bit header_received_on_A;
  ...
}
```

Figure 13: FRPocState CoverageObject Definition

The mechanism for initiating a functional coverage measurement on a CoverageObject is procedural, using the *coverObject()* method, rather than event driven. The main reason for this is to support coverage measurements which require multiple successive calls without advancing simulation time. For example, in this application we also need to perform coverage measurements on abstract data structures which are carried within a higher-level data structure, e.g. frames within a communication cycle; in this case multiple calls to *coverObject()* for different frames within the communication cycle are made without real time advancing. Implementing a generic solution using Vera events creates a complicated scheduling problem which does not exist with a simple procedural interface implementation. Some example code from a concrete implementation for the FRPocStateCov CoverageClass is shown in Figure 14.

```
class FRPocStateCov extends CoverageBase {

  protected event      cover_state_evt;
  protected event      cover_trig_evt;
  protected FRPocState poc;

  coverage_group poc_state_cov {
    sample_event = sync(ALL, cover_state_evt) async;
    sample poc.vState {
      // POC states (~18 statements)
      wildcard state WAKEUP_LISTEN (7'b010_xx01);
      ...
      // POC basic state transitions (~90 statements)
      trans DC_C ("DEFAULT_CONFIG" -> "CONFIG");
      ...
      // POC conditional state transitions (~170 statements)
      trans F02_15_004 ("NORMAL_PASSIVE" -> "HALT")
                       if ((poc.vCycleCounter[0] === 1)
                        && (poc.zSyncCalcResult === MISSING_TERM));
      ...
      // POC state transition sequences (~20 statements)
      trans POC_CSA_SCP ("READY" -> ... -> "NORMAL_ACTIVE");
      ...
    }
  }
  coverage_group poc_trig_cov {
    sample_event = sync(ALL, cover_trig_evt) async;
```

```
    sample poc.vState {
      ...
      // Trigger state coverage (~220 statements)
      state F07_3_005 (7'b010_xx01) if (poc.WUP_decoded_on_B === 1);
      ...
    }
  }
  task new(string name);
  virtual task coverObject(DataObjectBase obj);
  virtual function integer queryBin(string bin_pattern);
  virtual function integer queryHit(string bin_pattern);
  virtual task deactivateBin(string bin_pattern);
}
```

Figure 14: FRPocStateCov CoverageClass Definition

The CoverageClass operates on a single CoverageObject but has multiple *coverage_groups* with different requirements. Each *coverage_group* has a corresponding *sample_event*; in this case a corresponding local Vera event (e.g. *cover_state_evt*, *cover_trig_evt*) is triggered whenever the *coverObject()* method is called by the monitor. The *async* modifier is used with *sample_event* to ensure that successive calls to *coverObject()* which happen during the same simulator time-step all contribute to the overall coverage results. If the *async* modifier had been omitted only the CoverageObject information from the last call of such a sequence would contribute to the coverage.

Details of the state and transition *coverage_group* bin definitions were discussed in Section 4 of this paper; this section provides an overview of some of the concrete implementations of CoverageBase class methods. Figure 15 shows an example implementation of the *coverObject()* method. Note that the *cover_state_evt* is only triggered if a change of state has occurred; this is a requirement for using the state transition technique presented in this paper.

```
task FRPocStateCov::coverObject(DataObjectBase obj) {
  if (!cast_assign(this.poc, obj, CHECK)) {
    error(psprintf("cast_assign failed: %m"));
  }
  if (poc.change_of_state) {
    trigger(cover_state_evt); // only on change of state
  }
  trigger(cover_trig_evt);    // whenever coverObject() is called
}
```

Figure 15: Concrete Implementation of *coverObject()*

The *queryBin()*, *queryHit()* and *deactivateBin()* methods are provided to allow external components which instantiate the CoverageClass, i.e. CoverageMonitors, to access coverage information and control operation. The *queryBin()* method can be used to determine if the specified *bin_pattern* (which is a regular expression) exists in any of the *coverage_group* definitions for this CoverageClass. A CoverageMonitor which instantiates multiple CoverageClasses can use the corresponding *queryBin()* methods to determine if a *bin_pattern* exists in any of the associated *coverage_groups* before performing a high-level operation like *deactivateBin()*. An example implementation for *queryBin()* is shown in Figure 16.

```
function integer FRPocStateCov::queryBin(string bin_pattern) {
  if ((poc_state_cov.query(NUM_BIN, STATE|TRANS, bin_pattern) >= 1)
   || (poc_trig_cov.query (NUM_BIN, STATE|TRANS, bin_pattern) >= 1)) {
    debug(psprintf("%s is a valid bin name", bin_pattern));
    queryBin = 1;
  } else {
    debug(psprintf("%s is not a valid bin name", bin_pattern));
    queryBin = 0;
  }
}
```

Figure 16: Concrete Implementation of *queryBin()*

The *queryHit()* method can be used to determine if the specified *bin_pattern* has received a hit as shown in Figure 17. In this case the implementation must also check if the *bin_pattern* exists, since the predefined Vera *query()* function returns the value '0' for the *SUM* operator if the *bin_pattern* exists but has not been hit, or if a *bin_pattern* of that *bin_type* does not exist. In this implementation the *queryHit()* method uses a *fatal* to terminate the simulation if called with an illegal *bin_pattern*. Typically a CoverageMonitor would provide a higher-level public *queryHit()* method to allow constrained-random and directed testcases access to coverage results, as shown in Figure 19. In this case the CoverageMonitor hides the specific CoverageClass details from the testcase and uses appropriate *queryBin()* methods to determine which *coverage_group* contains the requested *bin_pattern*.

```
function integer FRPocStateCov::queryHit(string bin_pattern) {
  integer num_hits;
  if (queryBin(bin_pattern) == 1) {
    if (poc_state_cov.query(NUM_BIN,STATE|TRANS, bin_pattern) >= 1){
      num_hits = poc_state_cov.query(SUM, STATE|TRANS, bin_pattern);
    } else {
      num_hits = poc_trig_cov.query(SUM, STATE|TRANS, bin_pattern);
    }
  } else {
    fatal(psprintf("%s is an illegal bin_pattern", bin_pattern));
  }
  debug(psprintf("%s received %0d hit(s)", bin_pattern, num_hits));
  if (num_hits >= 1) {
    queryHit = 1;
  } else {
    queryHit = 0;
  }
}
```

Figure 17: Concrete Implementation of *queryHit()*

The *deactivateBin()* method can be used to set the *bin_activation* to *OFF* for all bins that match the specified *bin_pattern*; an example implementation is shown in Figure 18. Note that once the bin has been deactivated it no longer contributes to the coverage statistics, and more importantly it is removed from the data-base. If *queryBin()* is called on a bin that was successfully deactivated it will return a '0' to indicate that the *bin_pattern* does not exist and likewise a corresponding *queryHit()* operation would result in a *fatal*. So the *deactivateBin()* method does not have a corresponding *activateBin()*; all bins are active by default. The *deactivateBin()* method can be used to remove coverage points from the data-base that cannot be measured with a particular implementation due to inaccessibility of the required variables.

```
task FRPocStateCov::deactivateBin(string bin_pattern) {
  integer num_bins;
  if (queryBin(bin_pattern) == 1) {
    if (poc_state_cov.query(NUM_BIN, STATE, bin_pattern) >= 1) {
      num_bins = poc_state_cov.set_bin_activation(OFF, STATE, bin_pattern);
    } else if (...) {
      ...
    } else if (poc_trig_cov.query(NUM_BIN, TRANS, bin_pattern) >= 1) {
      num_bins = poc_trig_cov.set_bin_activation(OFF, TRANS, bin_pattern);
    } else if (...) {
      ...
    } else {
      fatal(psprintf("indeterminate bin_type for %s", bin_pattern));
    }
  } else {
    fatal(psprintf("%s is an illegal bin_pattern", bin_pattern));
  }
  warn(psprintf("bin_activation set to OFF for %0d bin(s)
                 matching bin_pattern %s", num_bins, bin_pattern));
}
```

Figure 18: Concrete Implementation of *deactivateBin()*

It is the responsibility of the CoverageMonitor to determine when an transaction is worth covering, at which point the monitor populates the CoverageObject and calls the appropriate concrete implementation of *coverObject()* from the corresponding CoverageClass. A CoverageMonitor is any Vera component that takes responsibility for calling *coverObject()*; so it may be a dedicated verification IP component or part of the responsibility of another monitor, checker or transactor. In our implementation for the FlexRay POC coverage the FRPocCovMon is the only component with a physical connection (via a virtual port) to all the signals required to construct the FRPocState CoverageObject. An excerpt from the FRPocCovMon CoverageMonitor is shown in Figure 19.

```
class FRPocCovMon extends MonitorBase {
  local FRPocCovPort    poc_port;
  local FRPocState      poc_obj;
  local FRPocStateCov   poc_cov;
  local integer         monitor_active;
  ...

  task new(string name, FRPocCovPort poc_port);
  task enable();
  task disable();
  function integer queryHit(string bin_pattern);

  protected task configure();
  protected task check();

  local task monitor();
  local task update_poc_obj();
  local task deactivate_void_bins();
  local function bit change_of_state();
  local function bit active_trigger();
  ...
}
// called once by sequencer at start of simulation
task FRPocCovMon::configure() {
  monitor_active = 1;
  deactivate_void_bins();
  fork {
```

```
    monitor();
  }
  join none
}
// continuously monitor for events worth covering
task FRPocCovMon::monitor() {
  while(monitor_active) {
    @(posedge poc_port.$Clk);
    if (change_of_state() || active_trigger()) {
      update_poc_obj();
      poc_cov.coverObject(poc_obj);
    }
  }
}
// called every time there is something to cover
task FRPocCovMon::update_poc_obj() {
  // example of enumerated type
  if (vera_is_bound(poc_port.$zSyncCalcResult)) {
    case (poc_port.$zSyncCalcResult) {
      2'b00 : poc_obj.zSyncCalcResult = WITHIN_BOUNDS;
      2'b01 : poc_obj.zSyncCalcResult = EXCEEDS_BOUNDS;
      2'b10 : poc_obj.zSyncCalcResult = MISSING_TERM;
    }
  }
  // example of boolean (enumerated) type
  if (vera_is_bound(poc_port.$zChannelIdleA)) {
    cast_assign(poc_obj.zChannelIdleA, poc_port.$zChannelIdleA);
  }
  // example of bit type
  if (vera_is_bound(poc_port.$header_received_on_A)) {
    poc_obj.header_received_on_A = poc_port.$header_received_on_A;
  }
  ...
}
// called once per simulation
task FRPocCovMon::deactivate_void_bins() {
  if (!vera_is_bound(poc_port.$vRemainingColdStartAttempts)) {
    poc_cov.deactivateBin("F07_13_010"); // an exact bin name
    poc_cov.deactivateBin("F07_13_011"); // all bins with this prefix
    ...
  }
  ...
}
...
```

Figure 19: FRPocCovMon CoverageMonitor Implementation

The port binding takes place in the top-level Vera program and this is the only place where implementation-specific decisions are made regarding the accessibility of parameters, signals and variables; for example in our implementation a few of the local SDL process variables are stored in RAM and not accessible to the Vera interfaces hence the corresponding port signals have void bindings. The monitor uses the *vera_is_bound()* system function to avoid making a run-time access to a port signal that has a void binding as shown in Figure 19. Unbound port signals have undefined values in the respective FRPocState CoverageObject fields. Since it is not possible to determine if coverage points that use these undefined variables have been hit they can be removed from the coverage data-base by calling *deactivateBin()* from the CoverageMonitor. An excerpt from the top-level Vera program is shown in Figure 20.

```
// Interface definitions (in-line or #included)
...
interface cov_if {
  input       clk             CLOCK       hdl_node "tb.top.clk";
  input [6:0] poc_state       PSAMPLE #-1 hdl_node "tb.top.poc_state";
  input [1:0] header_received PSAMPLE #-1 hdl_node "tb.top.core.header_rx";
  ...
}
// Port bindings (in-line or #included)
...
bind FRPocCovPort poc_cov_port {
  Clk                         cov_if.clk;
  vState                      cov_if.poc_state;
  header_received_on_A        cov_if.header_received[0];
  vRemainingColdStartAttempts  void;
  ...
}
...
program main {
  // declare all the verification components
  FRPocCovMon pocCovMonitor;
  ...
  // construct all the verification components
  pocCovMonitor = new("pocCovMonitor", poc_cov_port);
  ...
}
```

Figure 20: Top-Level Vera Program

6 Managing Coverage Data

FlexRay communication systems are highly configurable to support a large variety of network topologies and protocol operational characteristics. The FlexRay Protocol Specification defines more than 40 cluster parameters (which are valid for all nodes in the cluster, e.g. the number of slots in the static segment, the timing parameters for the communication cycle, etc.) and 30 node parameters (which may be different for each node in the cluster, e.g. if it is a startup node, which channels the node is connected to, etc.). The parameters values are not independent of one another and the specification defines the corresponding ranges, constraints and relationships. In order to claim conformance, implementations must support at least the specified range of configuration values identified in the protocol specification.

This layer of complexity over and above the basic protocol operation provides an additional challenge to measuring the functional coverage for a particular implementation. It is not sufficient to execute a regression and measure the coverage for a single configuration and yet it is not feasible to enumerate and measure the coverage for all possible configurations. In addition, for particular configurations not all of the coverage points are relevant, for example in a configuration where the DUT is a non-coldstart node it would not be capable of hitting all the coverage points related to startup operation.

Vera provides some simple mechanisms for merging and managing coverage databases which enables the coverage implementation to be isolated from these configuration dependencies. For each scenario in the regression suite the coverage database can be stored with a unique name which identifies both the testcase and an appropriate identifier for the corresponding set of configuration parameters (e.g. *testcase_id.config_id.db*). At the end of

the regression run the coverage databases can be interrogated to provide the required figures, typically:

- the overall coverage for the whole regression suite over all identified configurations
- the coverage for all testcases executed on a particular configuration set
- the relevant coverage for a particular testcase executed on all configurations

The first two sets of statistics can be achieved by executing *vera –cov_report* with appropriate filtering for the database filenames (e.g. **.db* for all results, **configX*.db* for *configX* results). In the third case many of the coverage points have to be rejected from the analysis. While this could be achieved from the testcase by deactivating all inappropriate bins it is a lot of effort to implement and maintain, a simpler solution is to generate the report for all configurations using the testcase name as a filter (e.g. **testY*.db*), and then to apply post-processing to the coverage report in order to select the relevant coverage points.

7 Conclusion

This paper presented a pragmatic solution to internal state coverage for POC operation and outlines its contribution to the overall functional coverage model of a FlexRay protocol engine. This coverage was successfully applied in a verification environment that supported both constrained-random generation and directed functional tests. In this application it was very difficult to reach all of the protocol corner cases using constrained-random stimulus and supplementary directed tests had to be defined; functional coverage analysis of the POC state transitions, conditional paths and triggers proved invaluable in identifying missing testcases and measuring progress and completeness of the regression test suite. The Vera skeleton code and architectural implementation, together with general background information, should prove useful to verification engineers facing similar functional coverage challenges in a variety of applications.

8 Acknowledgements

We would like to thank the following people for their contributions in reviewing the draft paper material and their general encouragement: *Jason Sprott* of Verilab, *Florian Bogenberger* and *Joachim Kruecken* of Freescale Semiconductor. In addition we would also like to thank the FlexRay Consortium for permission to reproduce the SDL diagrams for Figure 2 and Figure 3.

9 References

[1] FlexRay Consortium, Homepage, http://www.flexray-group.org
[2] FlexRay Consortium, FlexRay Communications System Protocol Specification, Version 2.0, 2004.
[3] SDL Forum Society, Homepage, http://www.sdl-forum.org/
[4] Synopsys Inc., Reference Verification Methodology User Guide.
[5] Synopsys Inc., OpenVera Language Reference Manual: Testbench.
[6] Allen Holub, UML Quick Reference, http://www.holub.com/goodies/uml/