

Poster Presentation DAC2009

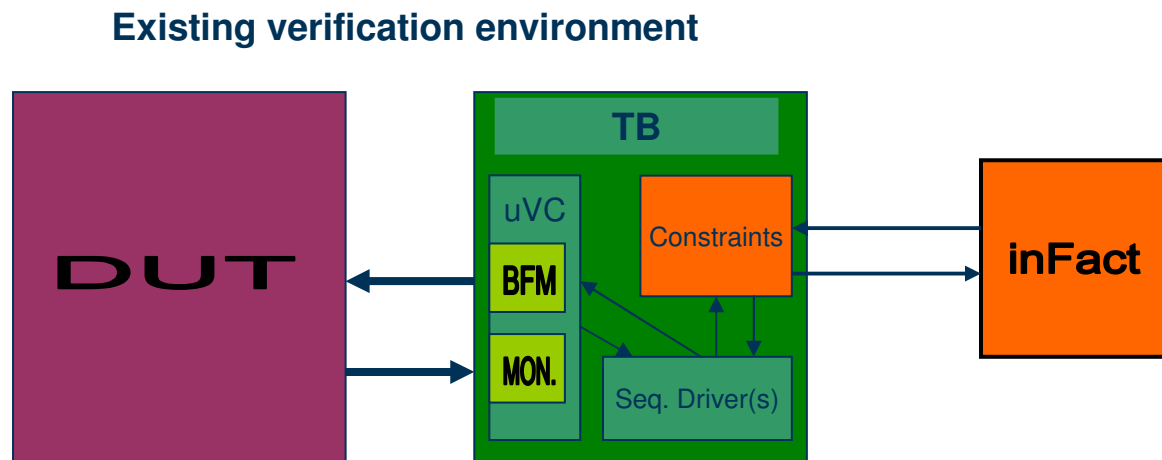
Using algorithmic test generation in a constrained random test environment

Author:

Håkan Askdal **ERICSSON** 

Overview

- An existing verification environment based on constrained random testing is extended with test generation using inFact
- The large investment in verification legacy in terms of uVCs, monitors, coverage, etc is kept - only the constraints are replaced



Background

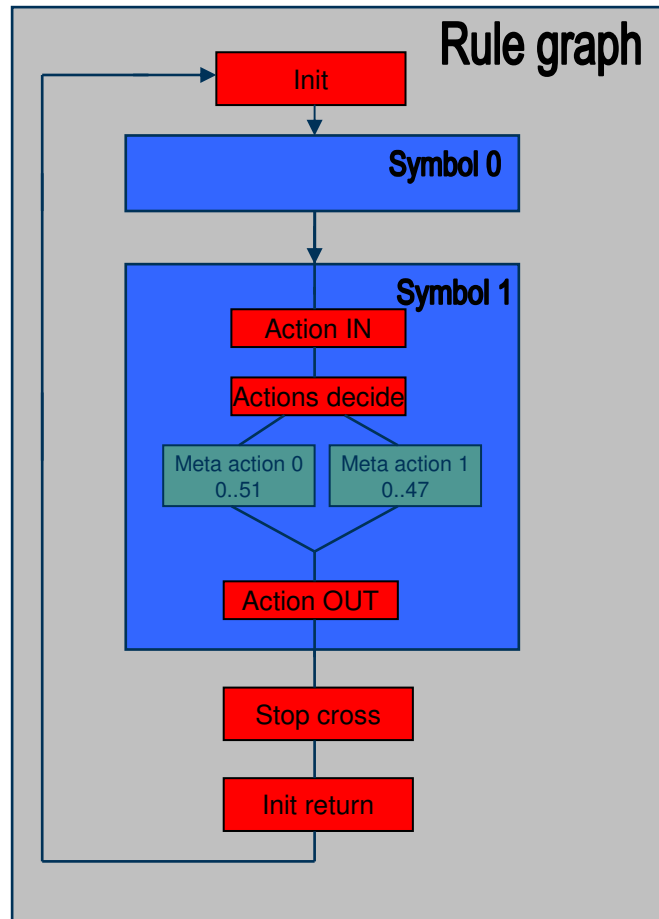
- Getting sufficient Functional Coverage in a timely manner is a challenging task in ASIC/SoC verification. For the last decade, logic simulation using **Constrained Random Stimuli Generation (CRSG)** has been considered as the state-of-the-art methodology. However, with growing complexity of designs, the size of the verification space tends to grow out of bounds for this technique.
- **Algorithmic Test Stimuli Generation (ATSG)** is a recent technology that promises to significantly shorten the time to closure. Using a rule-based approach to completely describe a protocol, a tool can then generate an exhaustive set of tests using a minimum number of patterns. Potentially, Algorithmic Test Generation can be orders of magnitudes more efficient than CRSG in terms of required number of simulation cycles.

What is inFact?

- An EDA-tool used for functional verification
 - Contain all components (scoreboard, checkers, test stimuli generation) to build proper test bench environment
- Algorithmic Test Stimuli Generation (ATSG)
 - Is the strong part in the tool which can systematically vary parameters to **generate as few test vectors as possible to obtain a full coverage goal!**
 - Is using its rule graph, BNF (Backus-Naur Form) in order to describe the possible test vectors
 - Path coverage is used to pick out unique vectors from the BNF
 - InFact has its own IDE (Integrated Development Environment) used for creation of the BNF

Rule graph

Architecture of an algorithmic test stimuli generator



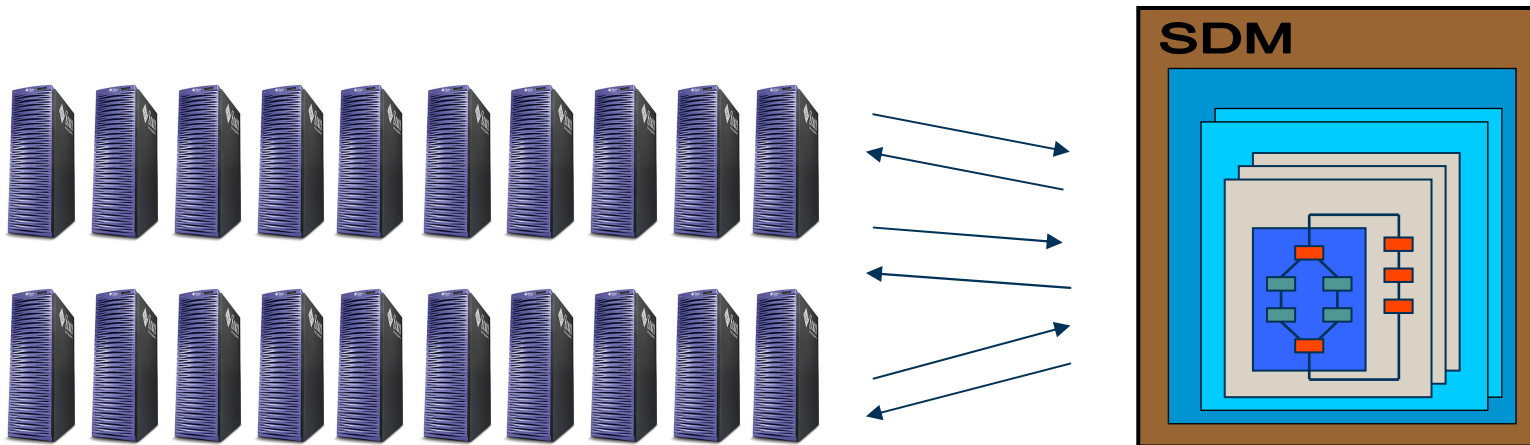
- The rule graph is described in a rule file and consists of one 'overall' BNF
- The 'overall' BNF consists of one or many symbols which are serially connected to each other. It also always includes three defined agents (init, stop-cross and init return)
- Each symbol contains one 'sub' BNF which consists of actions, meta actions and even other symbols
- The actions are used to routing the BNF and to connect symbols to other units in the rule graph
- From the meta actions values for stimuli generation are extracted out, from a defined range of values

Rule file

```
rule_graph DAC_2009 {  
    symbol symbol_1;  
  
    meta_action meta_action_0 [0..47];  
    meta_action meta_action_1 [0..51];  
  
    action init;  
    action stop_cross ;  
    action IN;  
    action OUT;  
    action decide;  
    action LEFT;  
    action RIGHT;  
    action init_return;  
  
    //Setup symbol  
    symbol_1 = IN decide (  
        ( LEFT meta_action_0 |  
          RIGHT meta_action_1)  
        ) OUT;  
  
    //Setup rule graph  
    DAC_2009 = init repeat {symbol_1 init_return  
        stop_cross};  
  
} // END rule_graph DAC_2009
```

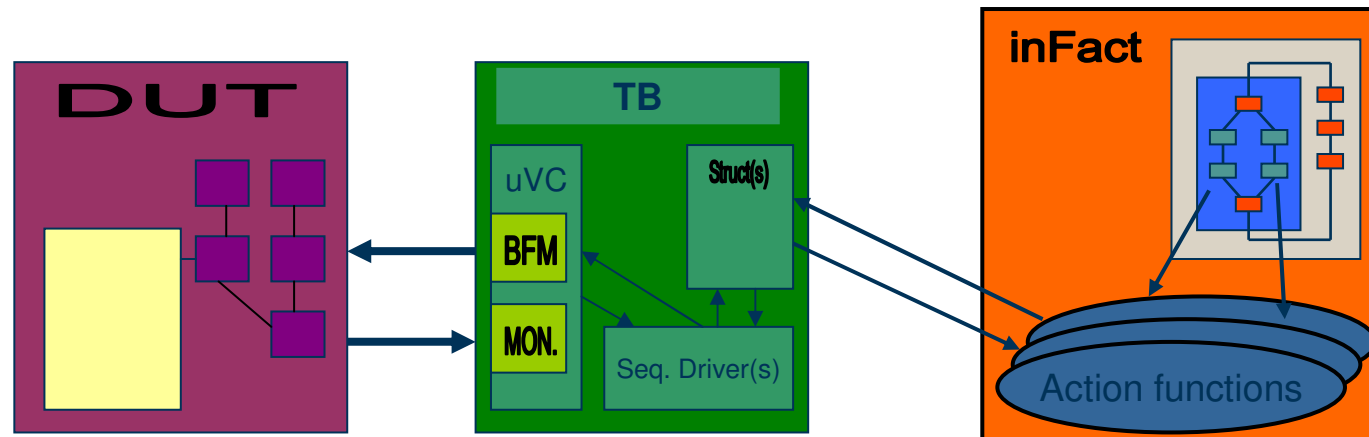
Simulation Distribution Manager (SDM)

- Is used for distribution of generated stimuli to a number of simulations from the same graphs
- SDM is running with its own process in parallel with the regression simulations
- The number of simulations to be run in parallel can be set
- A specific simulation can be replayed by using a file called DistributionHistory.out



Interface to the test bench's API

- inFact is connected to the Test Bench (TB) through the Application Interface (API) and the BNF can inside the TB be accessed through function calls
- Each symbol will have its specific function call which will update its specific struct



Interface to the TB's API

- 'e' code for connection to inFact

```

//*****
// FILE: top_inFact.e
//*****
C export conn_to_inFact;
C export initialization;

struct initialization {
    tx_offset_bfn          : uint(bits : 2);
    .
    .
    twelve_ant_IQB_transfer_length : uint(bits : 2);
};

unit conn_to_inFact {
    init_field : initialization;
    .
    .
}

prv_xio_driver_get_init(hdl_path:string,init:initialization):bool is foreign dynamic C \
    routine xio_engine:xio_engine_get_init;
};

get_graph_init( init : initialization)@sys.any is { //***** TCM!!!!*****
    while (!prv_xio_driver_get_init(me.hdl_path(), init)) {
        wait [1];
    };
};

// *****
// FILE: tc_inFact.e
// *****

extend sys {
    inFact : conn_to_inFact is instance;
};

extend MAIN sequence {

    graph : initialization;

    body() @driver.clock is only {
        .
        .
        // *****
        // FUNCTION CALL TO FETCH GRAPH VECTOR FROM INFACIT //
        // *****
        sys.inFact.get_graph_init(graph);
        .
        .
    };
};

```

- 'c++' code for connection to the TB

```

//*****
* FILE: <proj>_engine_impl.cpp
//*****

xio_engine_impl::xio_engine_impl()
{
    fTransactionState = 0;
    finitialization= SN_STRUCT_NEW(initialization);
}

void xio_engine_impl::tx_offset_bfn_action(int n) {
    finitialization->tx_offset_bfn = n;
}

//*****
* Called by the test bench in 'e' to retrieve the next request
//*****
sn_bool xio_engine_get_init(SN_STRING inst, SN_TYPE(initialization) init)
{
    printToLog("--> xio_engine_get_init()");

    AxFTestEngineVlog *eng_p = static_cast<AxFTestEngineVlog *>(findTestEngine(inst,
    "xio_engine"));

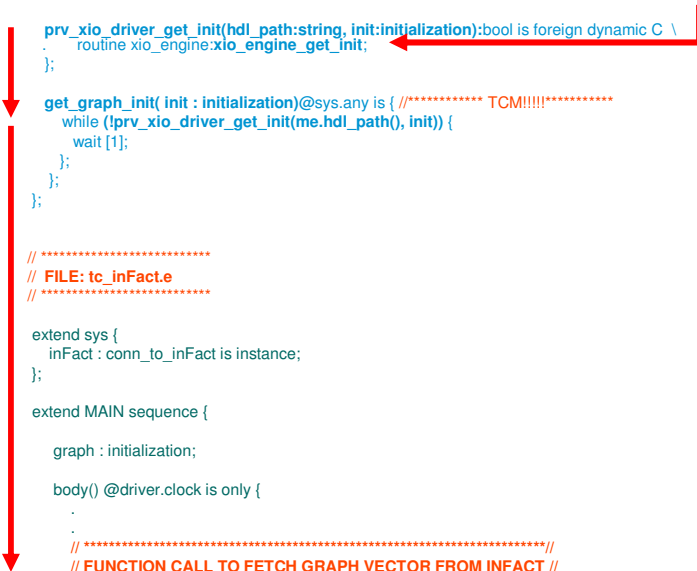
    xio_engine_impl *impl = static_cast<xio_engine_impl *>(eng_p->userData());
    if (impl && impl->getTransactionState() == TRANS_STATE_REQ_READY) {
        *init = *impl->getinitialization();
        // Wakeup the graph so it can continue execution
        printToLog("--> wakeup(init_available)");
        eng_p->wakeup("init_available");
        printToLog("<-- wakeup(init_available)");
    }

    impl->setTransactionState(TRANS_STATE_IDLE);

    printToLog("<-- xio_engine_get_init() -- true ");
    return true;
} else {

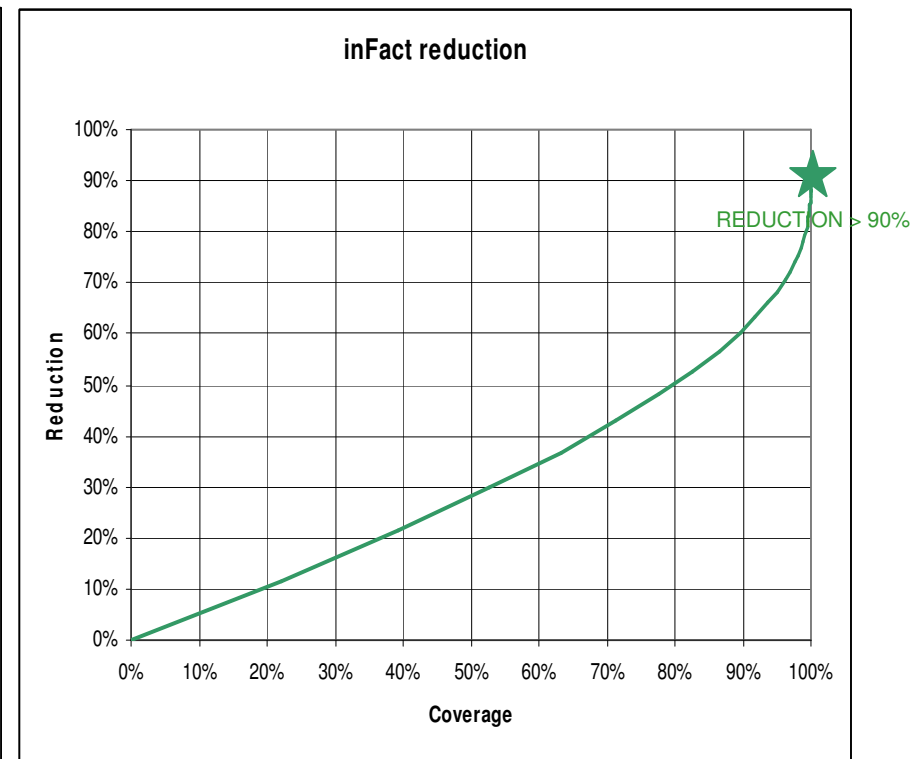
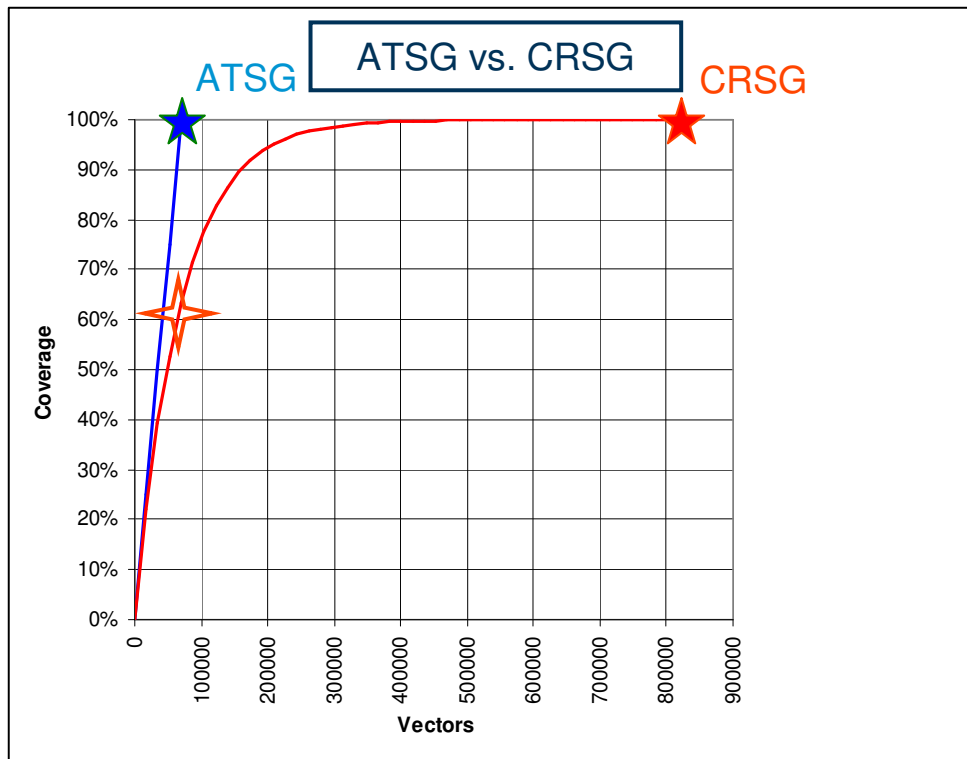
    //printToLog("NOT FINISHED!!!");
    printToLog("<-- xio_engine_get_init() -- false ");
    return false;
}
}

```



Results

- ATSG is in our example ten times more effective compared to CRSG



Results and conclusion

- InFact can be adapted into an existing TB through the API and replace constraints with rules from the rule graph
- Easier expression of similar test cases by merging and controlling several tests into the same rule file and just implementing one test case, which can be run in parallel using SDM in order to cover an exact number of vectors
- **Exceptionally faster reach of full coverage**