Post-Silicon Debugging of Many-Core Systems by Identifying Execution Paths Through Constraint Refinement

Amir Masoud Gharehbaghi

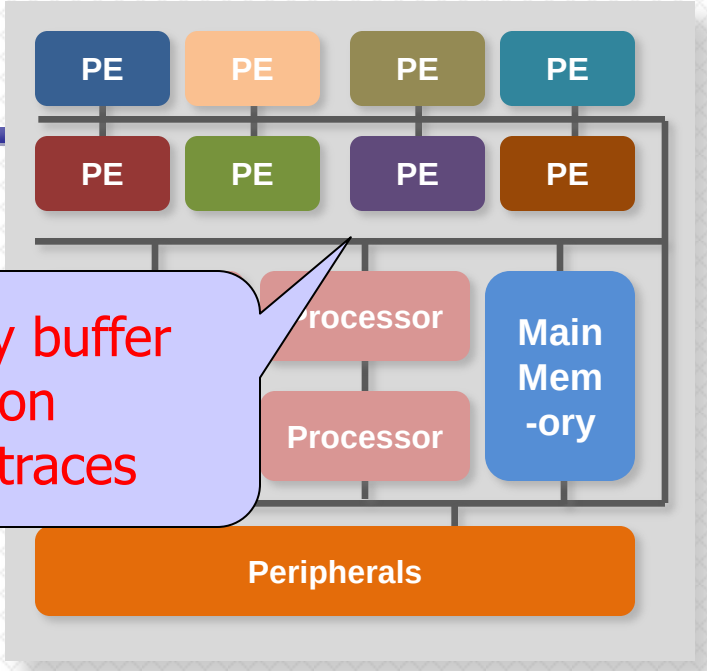
and **Masahiro Fujita**

VLSI Design and Education Center (VDEC), University of Tokyo

CREST, Japan Science and Technology (JST)

Summary

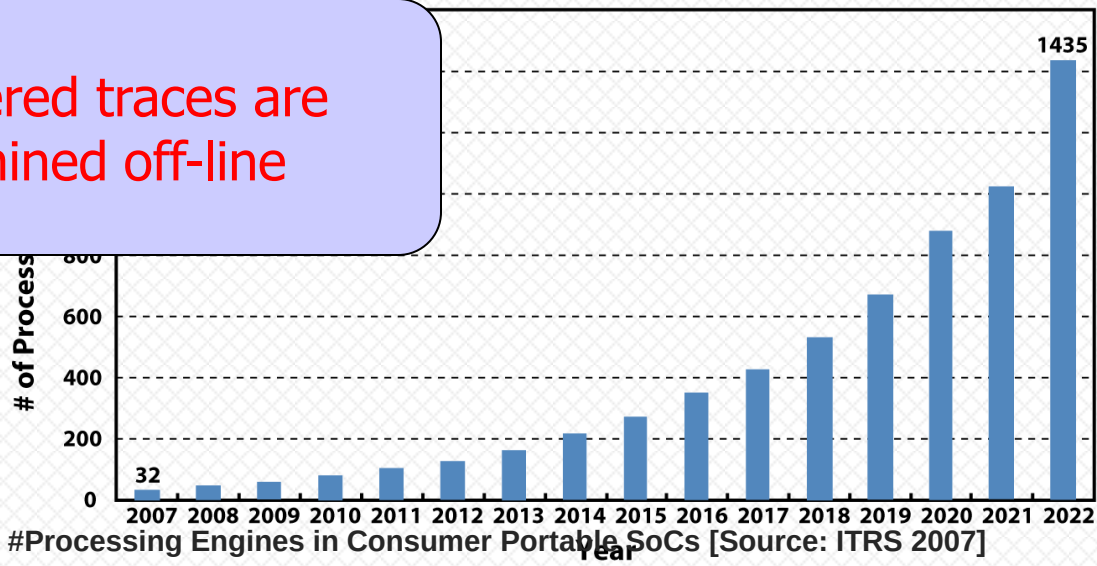
- Targeting post-silicon debug for SoC
- Use tracer with buffers for communication and also transaction level models
  - This give a set of potential execution paths
- Buffered traces are examined off-line with BMC (ESBMC)
  - Try to extract the real execution path



Automatically buffer communication /transaction traces

Buffered traces are examined off-line

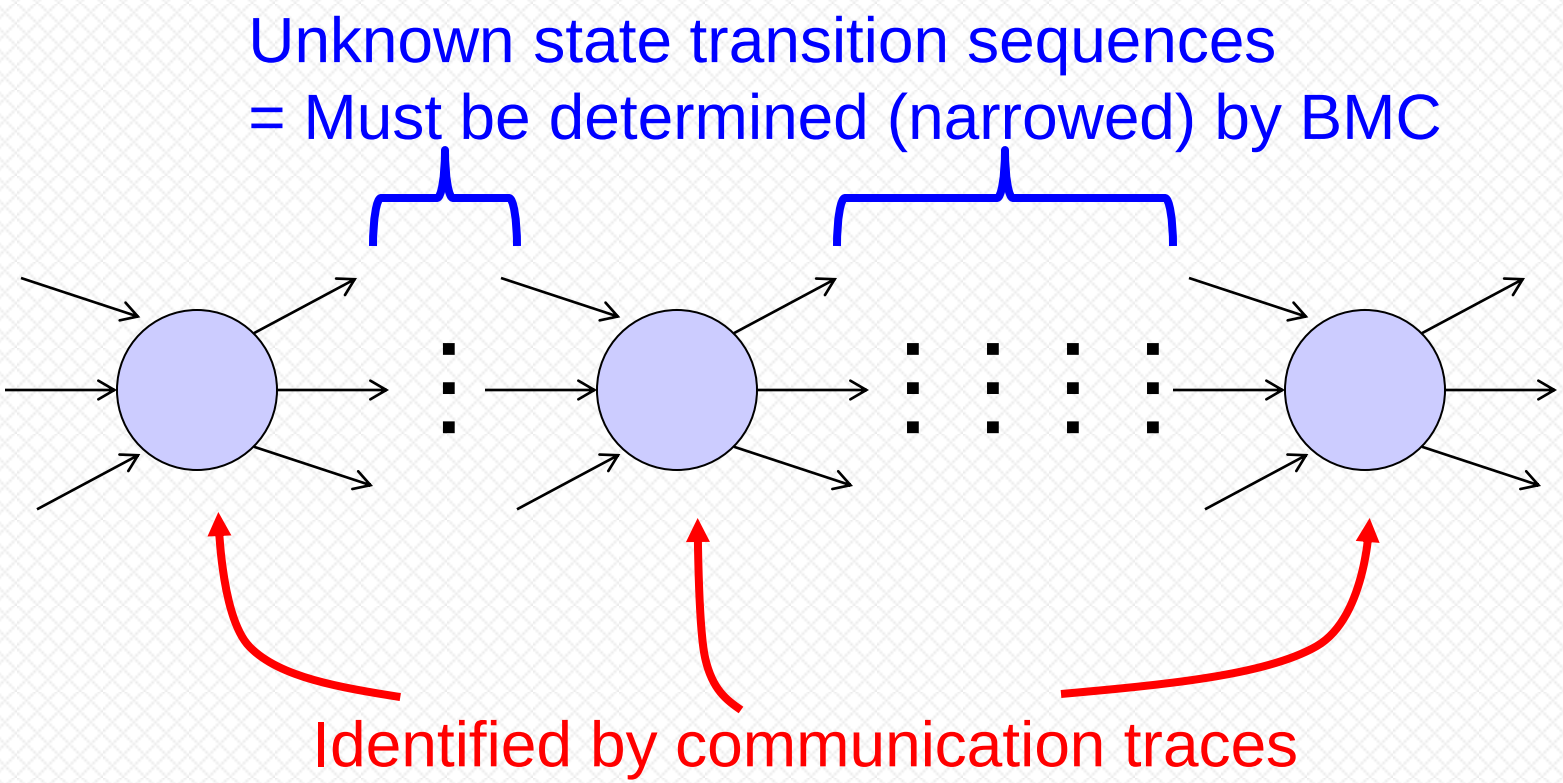
Con ortable SoC Template [Source: ITRS 2007]



#Processing Engines in Consumer Portable SoCs [Source: ITRS 2007]

Extracted paths

- Traces represent multiple paths which share a set of states that do communications



- Use BMC (ESBMC) to examine the above

Outline

- Introduction
- Overview of our method
- Transaction-level state machine
- Transaction-level backtracking and debug
  - Path generation
  - Bug localization
- Experimental results
- Conclusions

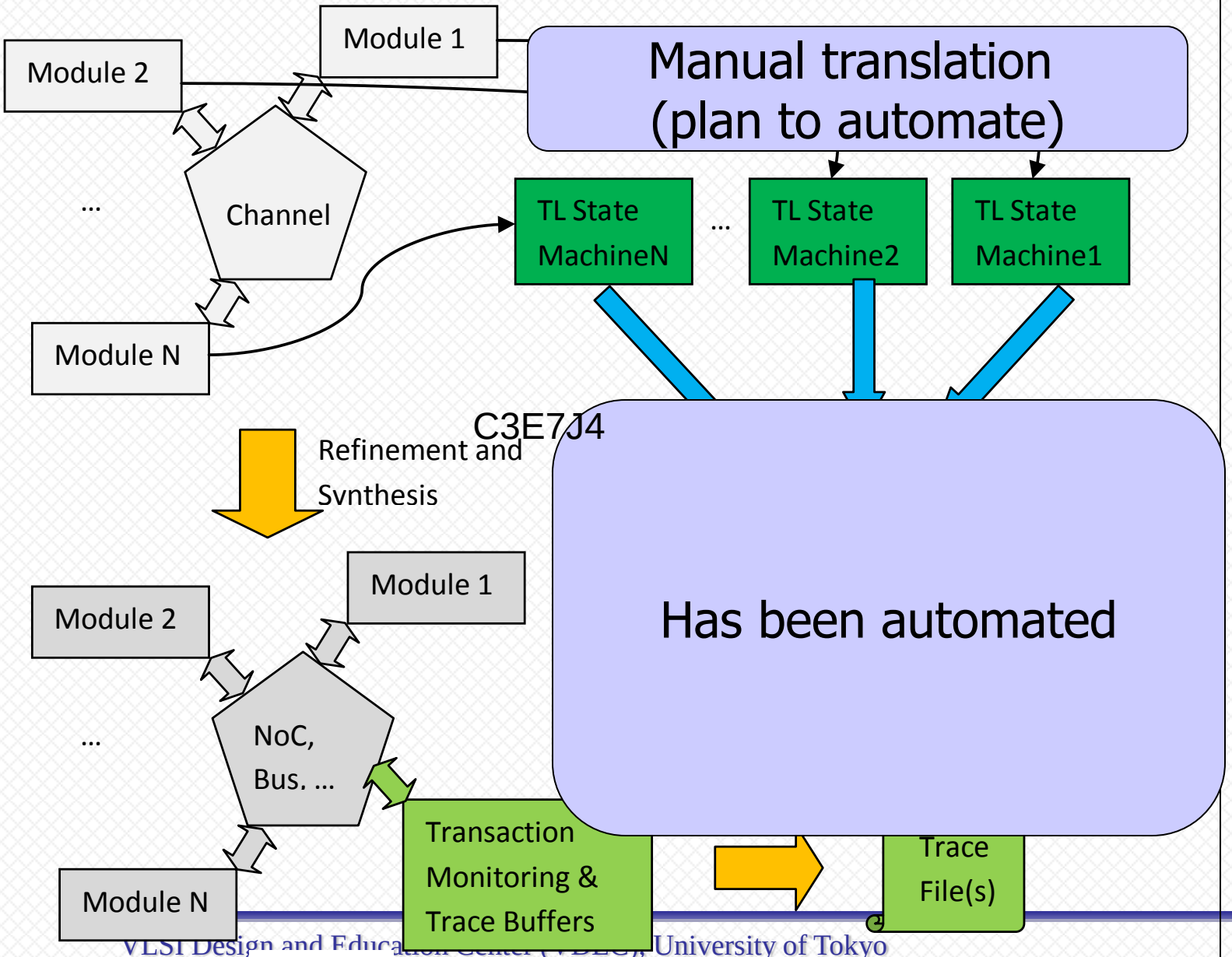
Introduction

- Complexity of modern SoCs is increasing
  - Number of cores is increasing (over 1,000 !)
  - Cores themselves maybe very complicated
- Communication among cores becomes more and more complicated
  - Multiple concurrent transactions
- Bugs may escape from pre-silicon to prototype or even to final system
- Post-Silicon debug is becoming a major task
  - Takes more than 50% of overall design time

Our Approach

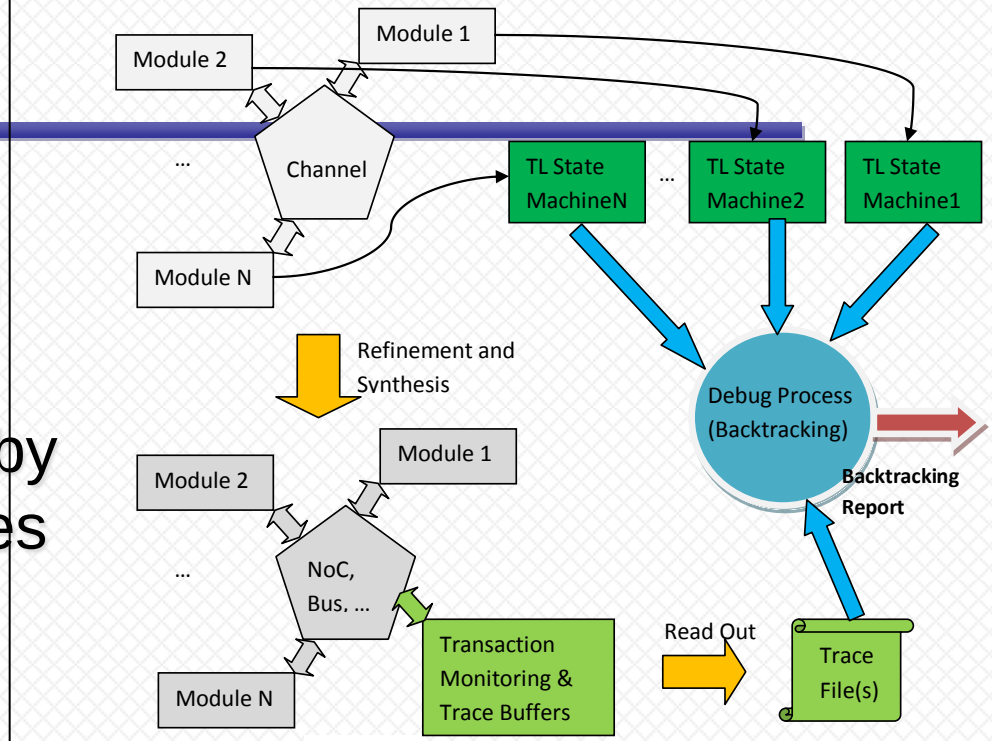
- Focus on functional bugs
- Consider communications among cores
  - Can be observed by monitoring communication channels (buses, NoCs, …)
- Link between chip transactions and high-level transactions
  - Assuming transaction-level design exists
  - Post-silicon debug with transaction-level analysis
- Backtrack in transaction-level design
  - Transaction-level path generation
  - Formal path analysis and constraint refinement

Overview of Our Method



Post-Silicon Debug Flow

1) Extract transaction-level behavior of modules from their TLM codes.
2) Instrument the hardware by adding monitoring modules and trace buffers to save transaction information during system operation.



3) Extract the transaction-level Run the system until a crash or failure state is reached or an error is detected.
4) Read the contents of the trace buffers and also the last state of the modules.
5) Run the debug process to backtrack in transaction-level states of the modules to find the bug(s).

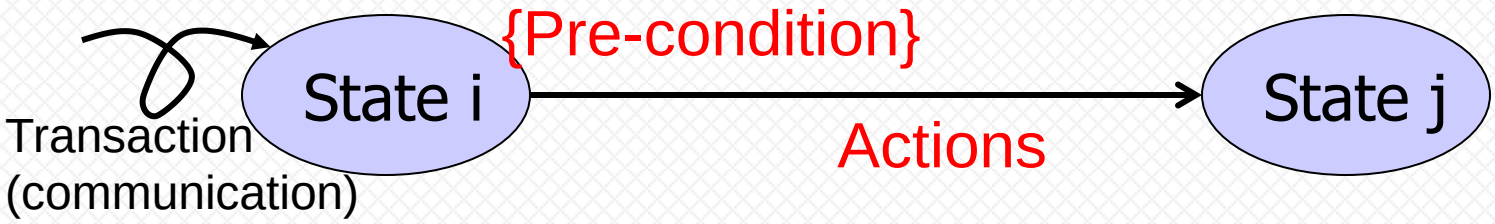
Transaction Monitoring

- Extract transaction information from signal events
  - General data
    - Initiator, target, command(read/write)
  - Application-specific data
- Monitoring circuit generation requires
  - Communication protocol
  - Application-specific data
- Trace buffer(s) contents

initiator ID	target ID	target Addr	Command	Data

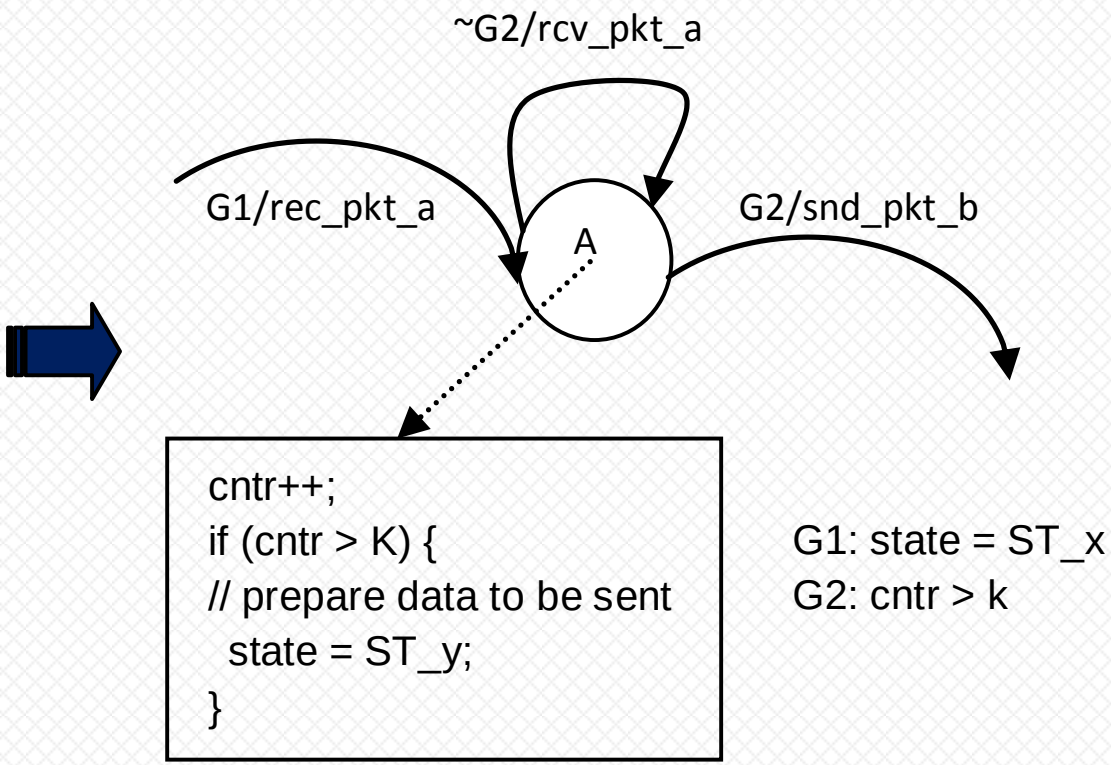
Transaction-Level State Machine

- For each module (core) extract one state machine
  - System consists of several concurrent state machines
- States correspond to high-level behavior of module
- Transition between states happens when a transaction is received and a pre-condition holds
- Pre-conditions (or guard expressions) only depend on internal variables/signals
- Transition between two states may result in an action that is initiating (sending) a transaction



TLSM Example

```
int cntr = 0;
While(1) {
  switch(state) {
  …
  case ST_x:
    wait_packet(pkt_i);
    if (pkt_i.type == PKT_ACK)
      cntr++;
    if (cntr > K) {
      state = ST_y;
     // prepare data to be sent
      send_packet(pkt_o);
    }
    break;
  case ST_y:
  …
  }
}
```



TLSM Formal Definition

- TLSM = (Ei, Eo,S, s0, G, T, A)

- Ei is set of input events (transactions)
- Eo is set of output events (transactions)
- S is set of states
- s0 is the initial state and belongs to S
- G is set of guard conditions
- T is transition function: Ei*G*S -> S
- A is action function: Ei*G*S -> Eo

TLSM Extraction

- Determine the functions for extraction process
  - Functions dealing with state variable(s) and also handling incoming and outgoing transactions
- Convert the TLM/SystemC/C++ code to a pure C code that represents the functionality dealing with state variable(s) and also the transactions
- Extract the TLSM states and their corresponding C code as a function
  - Abstract all the internal functionality
- Abstract the extracted code for the backtrack and analysis process

Some TLSM Extraction Details

- Some functionality of the modules are abstracted
  - User can define which parts to be abstracted
  - Some variable values may be replaced with new symbolic variables
    - All assignments to those variables are ignored
    - User may specify some constraint on the abstracted variables
  - Some functions may be abstracted as uninterpreted functions
    - All the code inside those functions are ignored
    - User may specify some constraint on return value of the abstrated functions

Some TLSM Extraction Details (2)

- C++ libraries (for example STL) are converted to their equivalent C codes
  - Also introducing bounds for some data structures such as array, list, … that can be unbound in their original form
- Considering specific coding style for using SystemC/TLM constructs to ease automation

Currently manual process (automatic program in development)
Need to decide which functions to be abstracted away

TLSM Extraction Example

```
case ST_RELEASING:

if (counter != locked_list_size) {
  packet.src_dest =
     locked_list[counter];
  packet.cmd = DL_FREE;
  packet.data = 0;
  send_packet(packet);
  counter++;
}
else {
```

```
if (ub_cond_size == 0) {
   state = ST_IDLE;
   counter = -1;
   succ_list_size = 0;
   pred_list_size = 0;
   locked_list_size = 0;
  }
  else {
   state = ST_WAIT_TO_LOCK;
   k = some_func1();
   counter = locked_list_size;
  }
  ub_cond_size = 0;
}
```

TLSM Extraction Example (2)

```
struct packet_info main_process_ST_RELEASING_simple();


struct packet_info main_process_ST_RELEASING_abs(char g1, char g2)
    {  // g1: mp_counter != locked_list_size  // g2: ub_cond_size == 0
  struct packet_info packet; packet.cmd = CMD_NONE;
  assert (state == ST_RELEASING);
  if (g1) {
    packet.src_dest = nondet_int();  packet.cmd = DL_FREE;
    packet.data = 0; }
  else {
    if (g2) { state = ST_IDLE; }
    else { state = ST_WAIT_TO_LOCK; }
  }
  return packet;
}
```

Debug Process

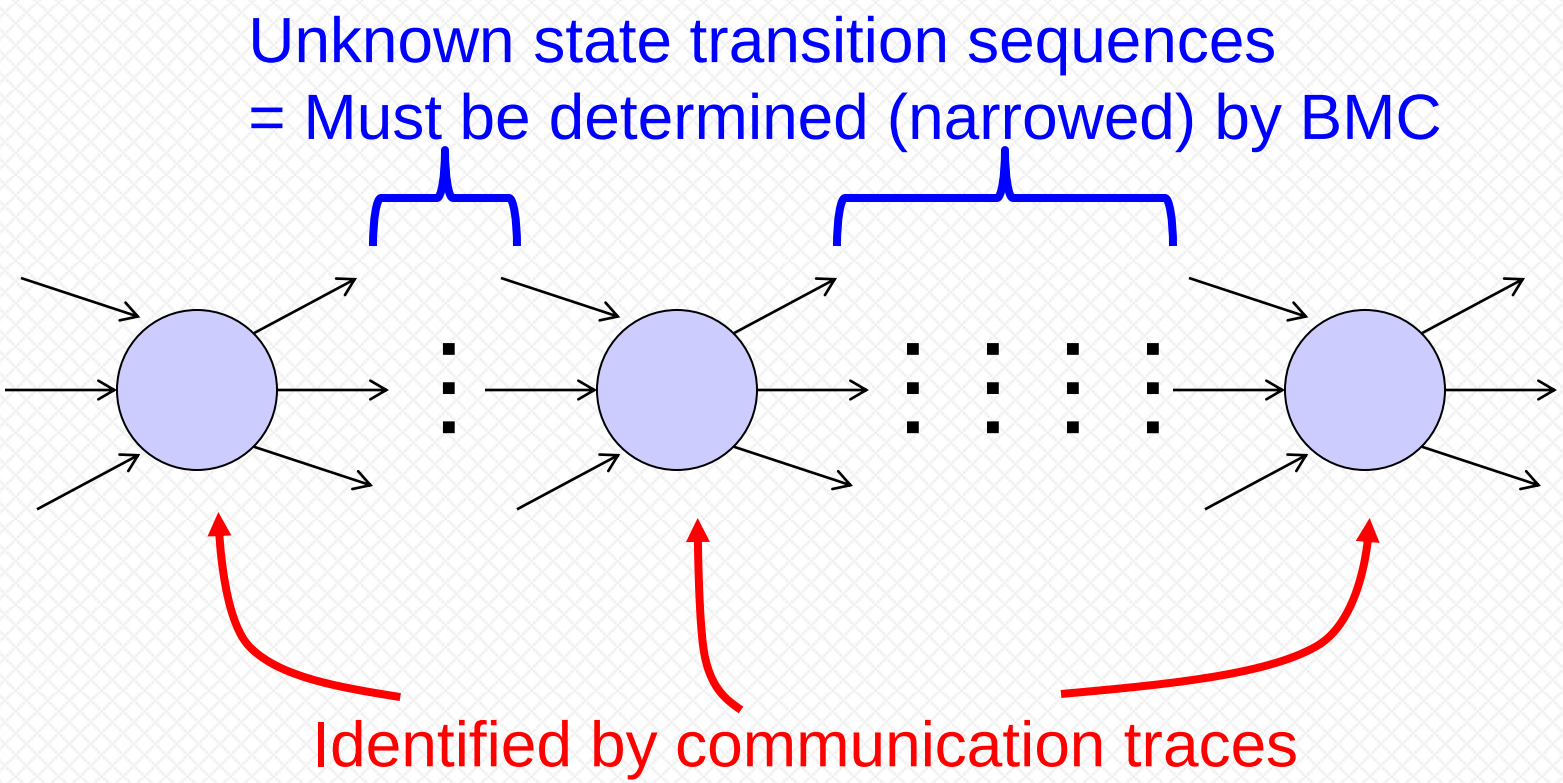
- Debugging is performed using:
  - The trace file
  - The extracted TLSM(s)
  - The last state of the target module(s)
- Two phase process
- Phase 1: path generation (for transaction-level backtracking)
  - Find bugs according to transaction behavior of the system
- Phase 2: path solver (bug localization)
  - Find cause of the bugs in more details according to the abstracted functionality

Path Generation

- Path generation is exercised using:
  - The trace file
  - The extracted TLSM(s)
  - The last state of the module(s)
- Beginning from the last state
- Following the observed transactions, find:
  - Possible (potential) previous states
  - Corresponding guard expressions
- Generate the output for path solver process to see if the path is actually feasible or not

Extracted paths

- Represent multiple paths which share a set of states



Path Generation Example

```
rec_rep_1.cmd = DL_REPORT;
rec_rep_2.cmd = DL_REPORT;
…
state = ST_DEADLOCK_DETECTION;
st_dd_g1_1 = 1;
st_dd_g1_2 = 1;
dl_wgh_g1_1 = nondet_uchar() % 2;
…
ret_00 =
    main_process_ST_DEADLOCK_DETECTION_abs(st_dd_g1_1);
assert(ret_00.cmd == DL_CALL);
…
assert(state == ST_DEADLOCK_DETECTION);
```

Path Solver

- For each generated path consider the actual functionality to find the bug
- Using BMC to find the values of internal variables
- Interactive process
- User should specify
  - Start and end of path (length of path)
  - Constraints on internal variables and the possible initial values (if known!)
  - Additional assertions to be checked
- Abstraction of functionality is necessary because of limitations of BMC

Path Solver Example

```
tileID = 16;

state = nondet_uint() % (ST_DEADLOCK_RESOLUTION+1);
__ESBMC_assume(state >= ST_IDLE && state <=
    ST_DEADLOCK_RESOLUTION);

weight_up = nondet_uint();
weight_dn = nondet_uint();
__ESBMC_assume(weight_dn != 0);
__ESBMC_assume(weight_up <= weight_dn);
…
state = ST_DEADLOCK_DETECTION;
g_is_deadlock_detection_active = 1;
```

Path Solver Example (2)

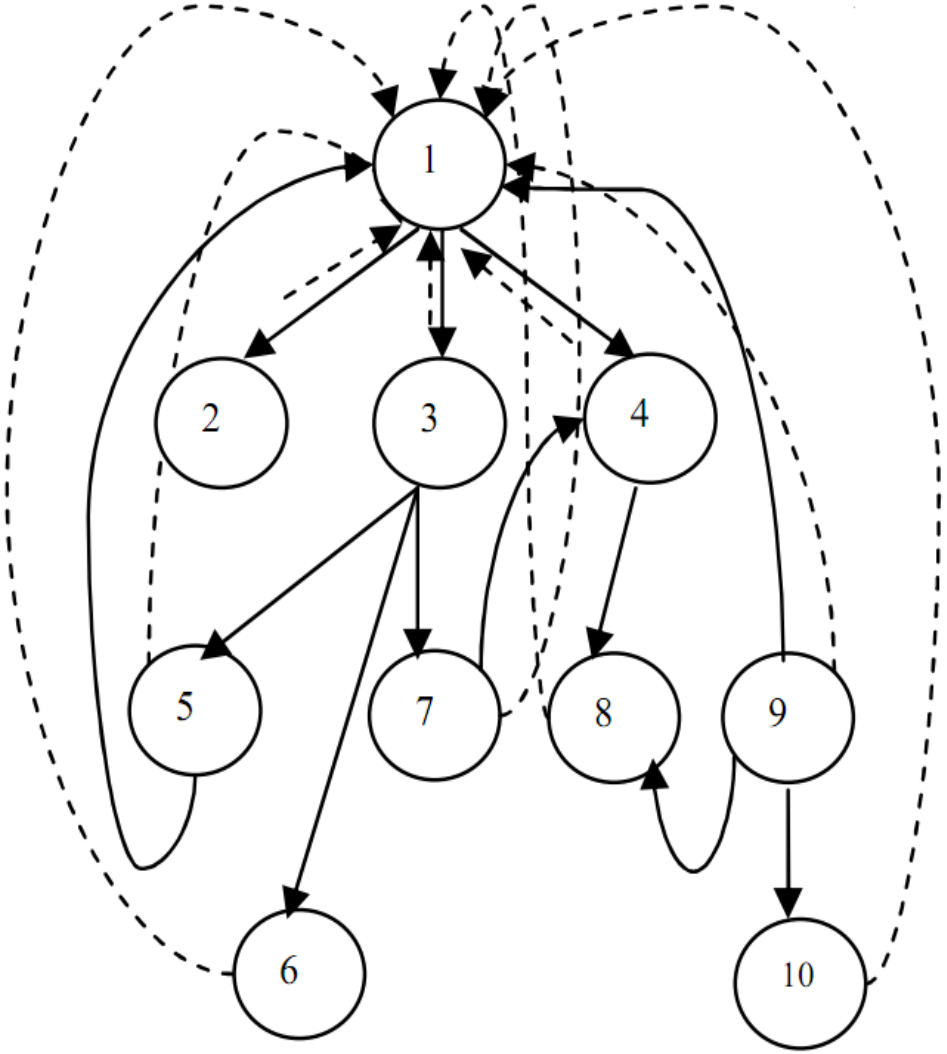
```
__ESBMC_assume(mp_counter != succ_list_size);
ret_00 =
   main_process_ST_DEADLOCK_DETECTION_simple(rand
   om_val_00);
assert(ret_00.cmd == DL_CALL);
__ESBMC_assume(ret_00.src_dest == 11);
__ESBMC_assume(ret_00.data == 268439554);
…
assert(weight_dn != weight_up);
…
assert(state != ST_DEADLOCK_DETECTION);
```

Case Study

- A Distributed Deadlock Detection and Resolution algorithm
- Several modules access shared resources
- Each module locks its required resources, does some (dummy) operation and releases them
- If locking is unsuccessful, a deadlock may have happened
- One of the modules begin deadlock detection and resolution

Deadlock Detection Overview

- Node 1 is the initiator of the detection and resolution process
- Solid lines represent query command to ask locked resources of each core
- Dashed lines show responses that are sent from each core to the initiator node (node 1)
- Finally, core 1 can determine from information from all other cores whether a deadlock is happened and how to (efficiently) resolve it



TLSM of Case Study

- Three concurrent processes are considered for TLSM extraction
  - Controlling state machine process
  - Lock/free handling process
  - Deadlock detection/handling process
  - Overall about 620 lines of code
- TLSM consists of:
  - 8 states
  - 121 transitions
  - 22 different guard expressions

Experimental Setup

- Modules with deadlock detection and resolution capability are implemented at transaction level
- Whole system consists of 25 modules in a 5*5 mesh NoC
- Nirgam NoC simulator is used for the network
- Whole system is simulated for 1000 cycles and transactions are logged during simulation
- Path generation process is implemented as a C++ program
- For bug localization, the ESBMC tool is used as our BMC engine and Z3 as SMT solver

Description of Bugs (1)

- Bug 1:
  - When the release request is sent to the module that has started the deadlock detection, it did not work.
- Incorrect sequence of transactions (observed and used in the path generation process):
  - sequence of sending and receiving release request by the module that has started the deadlock detection and resolution
- Constraint found during path analysis:
  - after sending the release command, the state does not change to the state for releasing the resources
- Cause of the bug:
  - This case has not been implemented

Description of Bugs (2)

- Bug 2:
  - beginning the deadlock resolution process before getting all the required information from other modules.
- Incorrect sequence of transactions (observed and used in the path generation process):
  - receiving a deadlock query response from a module after sending a deadlock resolution command to another module
- Constraint found during path analysis:
  - One of the internal variables for detecting the end of process becomes incorrectly more than 1
- Cause of the bug:
  - 2 responses are received incorrectly from a module

Description of Bugs (3)

- Bug 3:
  - not beginning the deadlock resolution procedure.
- Incorrect sequence of transactions (observed and used in the path generation process):
  - one of the modules does not respond to the initiator module (a missing transaction)
- Constraint found during path analysis:
  - overflow of one of the internal lists!
- Cause of the bug:
  - A problem in the implementation of the network!

Experimental Results

Bug	Transaction-level debug			C-based BMC		SMT solver			
	#modules	#messages	#paths	#assigns	#vcc	#funcs	#preds	#assumes	Runtime (s)
1	1	13	1	1014	31	273	737	1228	1.5
2	2	21	2	1797	47	483	1288	2156	2.9
3	5	31	2	3791	70	994	2597	4384	7.9

Conclusions

- We have presented a transaction-level post-silicon debug method that employs:
  - Transaction-level design information
  - Transaction information that is extracted during the system run
- We have introduced the transaction-level state machine and a mechanism to backtrack in transaction-level states
- Furthermore, we have used a mechanism to find constraints on internal variables of modules to pinpoint the bug

Thank You!

Questions?