Automatic Generation of Thread Communication Graphs from SystemC Source Code

SCOPES 2016, Germany

Tim Schmidt, Guantao Liu, Rainer Dömer

{schmidt.t, guantaol, doemer}@uci.edu

Center for Embedded and Cyber-Physical Systems
University of California, Irvine

The Adventure of Reusing 3rd Party Code

SystemC is a C++ library to model and simulate hardware and software components

A wait call synchronizes with the simulation kernel.
The Adventure of Reusing 3rd Party Code

- 1000s lines of code
- Shared variables
- References
- Nested control flow
- ... ???

- Illustrated module hierarchy
- Explicit control flow
- Resolved variable dependencies
- A picture tells a 1000 words
- ... !!!

Problem Definition

- Understanding and extending 3rd party source code is a lengthy, tedious, and hard process

- Graphical charts are helpful for the understanding of the
  - Hierarchical Structure
  - Multi threading
  - Communication Behavior

- Challenge
  - Available tools can recognize C++ constructs
  - Available tools cannot recognize SystemC semantics
  - C++ compiler is SystemC-agnostic
Outline

- Introduction and Motivation
  - Related Work
- Thread Communication Graphs
- RISC compiler
- Segment Graph
- Experimental Evaluation
  - Accuracy
  - Features and Limitations
- Conclusion

Related Work

[1] **CARH** is a project to generate a XML representation via Doxygen and other 3rd party tools.
(Hiren D. Patel, Deepak A. Mathaikutty, David Berner, and Sandeep K. Shukla)

[2] **Systemc-clang** analyzes statically communication properties.
   For TLM 2.0 models aspects like the socket name will be identified.
   (Anirudh Kaushik and Hiren D. Patel)

[3] **PinaVM** uses LLVM to extract structural information.
   (Kevin Marquet and Matthieu Moy)

[4] The **Segment Graph** is used for parallel simulation and race condition analysis in the context of SpecC.
   (Weiwei Chen, Xu Han, Che-Wei Chang, Guantao Liu, and Rainer Dömer)
Thread Communication Graphs

Plain functions

Notification dependencies

Thread Communication Graph

RISC: A Dedicated SystemC Compiler

- **RISC Software Stack**
  - Recoding Infrastructure for SystemC
    - C/C++ foundation
    - ROSE compiler infrastructure

- **ROSE Internal Representation**
- **Explicit support for**
  - Source code analysis
  - Source-to-source transformations
RISC: A Dedicated SystemC Compiler

- **RISC Software Stack**
  - Recoding Infrastructure for SystemC
  - SystemC Internal Representation
- **Class hierarchy to represent SystemC objects**

```
1) Segment Graph construction
2) Thread Communication Graph construction
```

```
Seg A: 1: a
       2: b
       3: e1.notify()

Seg B: wait(e2)

Seg C: 5: c
       6: d

Seg D: wait(e1)
       2: x
       3: y
       4: z
       5: e2.notify()
```

(c) 2016 T. Schmidt, CECS
RISC: A Dedicated SystemC Compiler

- **Segment Graph**
  - Segment Graph is a directed graph
  - Nodes: Segments
    - Code statements executed between two scheduling steps
      - Expression statements
      - Control flow statements (if, while, ...)
      - Function calls
  - Edges: Segment transitions
    - Primitives that trigger scheduler entry
      - wait(event)
      - wait(time)
  - Segment Graph can be constructed statically by the compiler from the model source code
    - (see example on next slide)
RISC: A Dedicated SystemC Compiler

- Segment Graph Construction:
  - Support for loop statements
    - while, do-while, for (with break, continue)
    ```c++
    void while_continue_statement()
    {
      int kk;
      while (test) {
        int aa;
        wait();
        int bb;
        if (test1) {
          continue;
        }
        int oo;
        wait();
        int cc;
      }
      int dd;
      wait();
    }
    ```

- Segment Graph Construction:
  - Support for function calls
    ```c++
    void f() int gl()
    {
      int aa; int g_0;
      wait(); wait();
      int bb; int g_1 = 33;
      gl(); if (g_1 == 88) {
        int cc; int g_2;
        wait(); wait();
        int dd; int g_3 = 44;
        return 43;
      } else {
        int DEAD_CODE;
      }
      int g_4; wait();
      int g_5; wait();
      int g_6; return return_value;
    }
    ```
Thread Communication Graph

- Segment Notification Analysis:
  - Example:

```cpp
class Simple: public sc_module {
    void sender() {
        wait(request);
        acknowledge.notify(SC_ZERO_TIME);
        for(int i = 0; i < 10; i++) {
            send_data();
            event.notify(SC_ZERO_TIME);
            wait(SC_ZERO_TIME);
        }
        wait(SC_ZERO_TIME);
        end.notify(SC_ZERO_TIME);
        wait(SC_ZERO_TIME);
    }
    void receiver() {
        request.notify(SC_ZERO_TIME);
        wait(acknowledge);
        for(int i=0;i<10;i++) {
            wait(event);
            receive_data();
        }
        wait(end);
    }
};
```

---

Thread Communication Graph

- Segment Notification Analysis:
  - Example:

```cpp
Sender: main INIT
Segment ID: 1
Ack.notify(SC_ZERO_TIME)
send_data()
event.notify(SC_ZERO_TIME)
Segment ID: 4
send_data()
event.notify(SC_ZERO_TIME)
Segment ID: 5
End.notify(SC_ZERO_TIME)
Segment ID: 6

Receiver: main INIT
Segment ID: 8
Req.notify(SC_ZERO_TIME)
Segment ID: 10
Segment ID: 12
receive_data()
Segment ID: 12
compute_checksum()
```
Thread Communication Graph

- Segment Notification Analysis:
  - Example:

```
Sender::main INIT
Segment ID: 1
Ack.notify(SC_ZERO_TIME)
Send_data()
Event.notify(SC_ZERO_TIME)
Segment ID: 4
Send_data()
Event.notify(SC_ZERO_TIME)
Segment ID: 5
End.notify(SC_ZERO_TIME)
Segment ID: 6
```

```
Receiver::main INIT
Segment ID: 8
Req.notify(SC_ZERO_TIME)
Segment ID: 10
Receive_data()
Segment ID: 12
Compute_checksum()
```

Segment Notification Table
Experimental Evaluation

AMBA Bus example

Structural Hierarchy
Top
- Channel master1_to_decoder
- Channel master2_to_decoder
- Channel address_bus_to_decoder
- Channel master1_to_data_bus
- Channel master2_to_data_bus
- Channel data_bus_to_slave
- Channel decoder_to_slave
- Channel master1_to_slave_bwrite
- Master1 m1
- Master2 m2
- Arbiter arbiter
- DataBus databus
- Slave slave
- Decoder decoder

Thread Communication Graph

Accuracy of Static Analysis

- Static analysis can lead to false positive/negative situations
  - Communication can be misinterpreted and illustrate too many or too few edges
  - Loops are treated as never taken, once taken, or unrolled (as far as possible)

- Static analysis is very helpful for a first picture
  - Notify and wait dependencies can be followed the nested hierarchy
  - Designer can make annotations for the static analysis
Features and Limitations

Features:
- Resolving of references
- Selection of a sub tree in the module hierarchy
Features and Limitations

Features:
• Resolving of references
• Selection of a sub tree in the module hierarchy
• Support for pseudo comments for graph annotation

... void func()
|
...
wait(req_event); // my comment
...}
...

Features and Limitations

Features:
• Resolving of references
• Selection of a sub tree in the module hierarchy
• Support for pseudo comments for graph annotation
• Filtering of communication types
  • Events for synchronization
  • Message passing via channels
Features and Limitations

Features:
- Resolving of references
- Selection of a sub tree in the module hierarchy
- Support for pseudo comments for graph annotation
- Filtering of communication types
  - Events for synchronization
  - Message passing via channels
- Export to PDF files

Limitations:
- No support for pointers
- Limited support for arrays of ports
- Started adding dynamic analysis
- Charts do not show timing/ordering
Conclusion

- Analysis of 3rd party SystemC code is a lengthy, tedious, and hard process
- General purpose tools like Doxygen have limited support
- We introduced a dedicated SystemC compiler to generate automatically
  - print the structural hierarchy
  - graphical communication charts
- Demonstrated on an abstract model of an AMBA bus

Thank you very much for your attention.

Questions?