Automatic Generation of Thread Communication Graphs from SystemC Source Code

Tim Schmidt, Guantao Liu, and Rainer Dömer
Center for Embedded and Cyber-physical Systems
University of California, Irvine, USA
schmidtt@uci.edu, guantaol@uci.edu, doemer@uci.edu

ABSTRACT

In an ideal top-down system design flow, graphical diagrams are designed before an executable specification in a System Level Description Language (SLDL) is derived. Such initial charts typically also serve as visual documentation of the textual specification and aid in maintaining the model. In the absence of graphical charts, e.g. in case of legacy or 3rd party code, a textual SLDL model is hard to comprehend for any unfamiliar designer. Here, we propose to automatically extract graphical charts from given SystemC code to ease the understanding of the source code with a visual representation. Specifically, we extract the communication flow between the threads from the design model by use of an automatic SystemC compiler infrastructure that statically analyzes the code and generates custom Thread Communication Graphs (TCG) similar to message sequence charts. Our experimental results on embedded applications demonstrate that our novel static analysis can quickly extract accurate TCG that are highly useful for designers in becoming familiar with new source code.

1. INTRODUCTION

A picture is worth a thousand words. In the absence of a picture or graphical charts, the model of a system described in a System Level Description Language (SLDL) is hard to comprehend. For the system designer who faces legacy code that needs to be reused and revised, identifying essential design elements, such as the main design modules, threads and their communication patterns, becomes a tedious and lengthy task. Before adjustments, improvements or extensions of the model can be applied, the existing source code needs to be read, analyzed and fully understood. Without documented schematic charts, such reverse engineering can require months of unproductive time.

In this paper, we propose an automated technique to quickly generate graphical charts from SystemC source code that can help the system designer in understanding third party or legacy models. Our automatic chart generator, which is based on sophisticated static code analysis by a novel SystemC compiler, quickly produces module hierarchy trees and multi-thread communication charts that assist the designer to puzzle out and grasp the intricacies of the complex source code. As such, this work provides a powerful resource in getting familiar with legacy or third-party SystemC code.

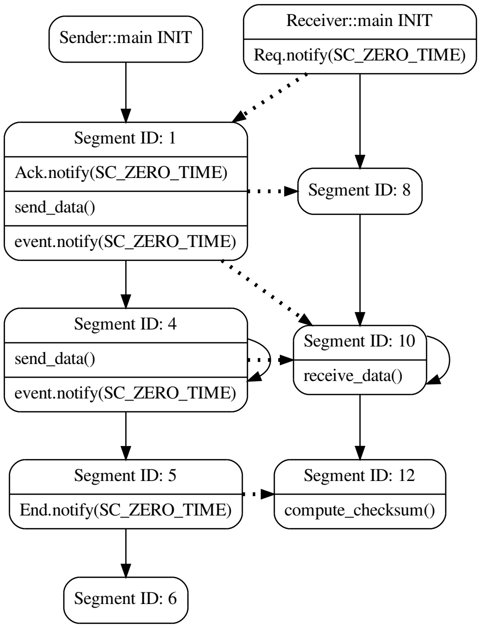


Figure 1: Generated Thread Communication Graph (TCG) from SystemC code

Figure 1 shows an example Thread Communication Graph (TCG) that we generated automatically from original SystemC source code for a model where a pair of modules obviously communicates in producer-consumer fashion. Instead of reading the 586 lines of source code distributed over 3 files that make up the design model, a simple glance over the quickly generated figure reveals the essential communication protocol used between the modules Sender and Receiver. A closer look at the chart then shows that events Req, Ack, and End are used to synchronize the communicating parties, then data is exchanged in a loop, and finally a checksum is computed. The automatically generated TCG clearly saves

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SCOPES '16, May 23-25, 2016, Sankt Goar, Germany
© 2016 ACM. ISBN 978-1-4503-4320-6/16/05…$15.00
DOI: http://dx.doi.org/10.1145/2906363.2906365

the designer precious time in understanding the model.

The key contributions of this paper are as follow:

- We developed a dedicated compiler to analyze the structural and behavioral aspects of legacy SystemC code.
- We integrated the SystemC discrete event simulation semantics in the analysis.
- We designed a novel algorithm to extract communication patterns and illustrate them in form of a chart.
- We demonstrated the capabilities of our project on a 3rd party library and an abstract AMBA bus model.
- We contribute our SystemC compiler to the open source community.

1.1 Related work

Static analysis of source code has been discussed in various works. CARH [1] is an architecture for validating system-level designs. The software documentation generation tool Doxygen [2] and other open source tools generate a XML representation of source code. Our work differs in that we analyze and identify the structural and behavioral aspects of the model. Specifically, we focus on the communication pattern in a given design. We utilize the knowledge of Discrete Event Simulation (DES) semantics to achieve a deeper recognition of the design. In comparison, general purpose tools like Doxygen cannot handle this task. The missing sensitivity to the SystemC semantics does not allow deeper analysis. Doxygen is familiar with C++ constructs like classes, templates, functions, and other concepts. However, the tool is not trained to analyze modules, channels, and event notifications and cannot extract communication.

A very similar tool is the systemc-clang framework [3] for static analysis of SystemC models which generates an intermediate representation of RTL and TLM designs. systemc-clang can identify communication properties (callback function name, socket name, payload information, ...) through static analysis in TLM 2.0 style. Their compiler recognizes communication function calls. The attributes of the communication type are analyzed through the function parameters. However, we are analyzing the port binding, linked channels, and communication peers as well. Thus, our approach is designed for static analysis of structural and behavioral communication charts among the threads in the model.

PinaVM [4] is a tool which bases on LLVM to extract structural information. This work is inspired from [5]. Scoot [6] is a tool for type checking to gain faster simulation via code re-synthesis.

A Segment Graph was first proposed in [7] for out-of-order scheduling to speedup simulation in context of the SpecC language, and later [8] to detect race conditions and parallel execution conflicts. In contrast, we are generalizing the concept of the Segment Graph for SystemC and aim at extracting communication graphs from source code.

The rest of this paper organized as following. We introduce the TCG in Section 2. Following, in Section 3 we explain how to generate a TCG from a Segment Graph. Our experiments are presented in Section 4, followed by a summary and future work in Section 5.

2. THREAD COMMUNICATION GRAPH

The design process of embedded systems typically requires combining various in-house modules and third party components, each of which the system designer needs to become familiar with. Minimizing the time to study new parts is critical, so the designer needs to focus on the essential aspects, including the communication and causal chain of composed components. Tools like Doxygen can generate call graphs to illustrate the software function hierarchy. However, these tools are agnostic to the system design semantics. To be effective, SystemC constructs for structure and communication need to be recognized and properly represented.

The problem of missing semantic analysis is illustrated in Figure 2. Two functions are shown in Figure 2a, independent entities, as software tools would see them. When SystemC semantics are applied, threads and events can be identified and the chart in Figure 2b can visualize the synchronization between the two threads. Even more so, we can utilize the knowledge of Discrete Event Simulation (DES) semantics and explicitly show the communication and timing dependencies between the two threads.

Figure 2c shows the resulting graph when `wait()` is correctly represented as a construct that incurs a delay due to the underlying multi-thread scheduling. For the remainder of this paper, we will refer to this as a *segment boundary* that separates the *segments* of code that SystemC semantics imply as being executed without interruption. We will formally define the corresponding *Segment Graph* below in Section 3.

In Figure 2c, the solid edges show the flow of thread execution over individual segments, whereas the event notification dependencies are indicated by dashed lines. Here, segment A notifies event `e1` for which segment D is waiting. Thus, the designer can see immediately that segment D must be executed before segment B.

Please note that for this analysis references and port mappings of events must be resolved as well. For this, our approach generates an instance tree over the entire design hierarchy so that shared variables can be correctly disambiguated.

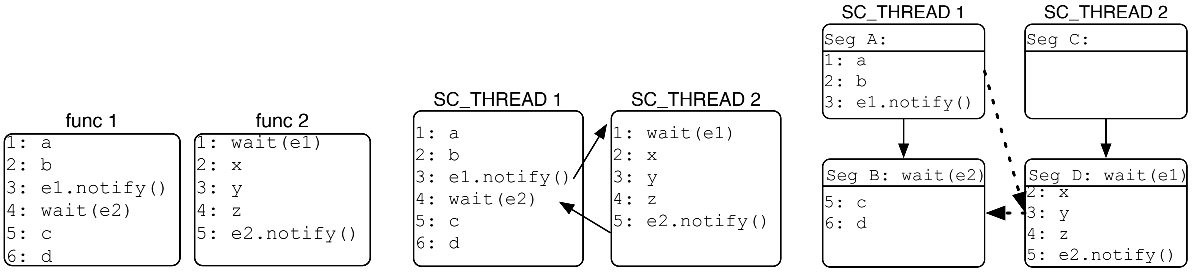
3. STATIC COMPILER ANALYSIS

At the core of our SystemC visualization is the Recoding Infrastructure for SystemC (RISC), an advanced compiler framework for analyzing, executing and instrumenting SystemC models.

3.1 RISC Compiler Infrastructure

Figure 3 shows the software stack of RISC. On top of a C/C++ software foundation, we selected the ROSE compiler [9] and its internal representation (IR) to generate and maintain an abstract syntax tree (AST) of the design model and the SystemC library. Our SystemC IR layer represents SystemC constructs, such as modules, channels, instances, threads and ports, explicitly (similar to [3]). On top of this, we have placed our Segment Graph Generator which analyses and visualizes thread execution and synchronization.

Our Segment Graph Generator is a parameterizable algorithm to extract segments of a given scope for a program (e.g. function `foo` in Figure 4a). Formally, a *Segment Graph* [7] consists of *segment* nodes and transition edges. A segment includes all reachable code from a *segment boundary*. Segment boundaries are generally user defined and can



```
1: a
2: b
3: e1.notify()
4: wait(e2)
5: c
6: d
```

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

(a) Two functions

```
1: a
2: b
3: e1.notify()
4: wait(e2)
5: c
6: d
```

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

(b) Threads and events

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

```
Seg B: wait(e2)
5: c
6: d
```

```
Seg C:
```

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

(c) DES dependency

Figure 2: Analysis of communicating threads

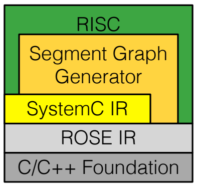
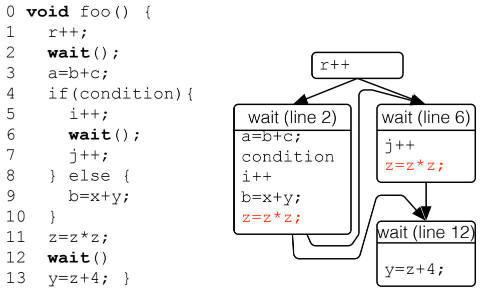


Figure 3: Software Stack



```
0 void foo() {
1   r++;
2   wait();
3   a=b+c;
4   if(condition){
5     i++;
6     wait();
7     j++;
8   } else {
9     b=x+y;
10  }
11  z=z*z;
12  wait()
13  y=z+4; }
```

(a) Source Code

```
r++
```

```
wait (line 2)
a=b+c;
condition
i++
b=x+y;
z=z*z;
```

```
wait (line 6)
j++
z=z*z;
```

```
wait (line 12)
y=z+4;
```

(b) Segment Graph[wait]

Figure 4: Recoding Infrastructure for SystemC (RISC)

be any functions calls or control flow statements (e.g. **if**, **while**, ...). Here, we use **wait()** as boundary, since we are interested in the flow of time with synchronization dependencies.

The edges in a Segment Graph are defined by control flow transitions. A transition exists between two segments S_1 and S_2 if segment S_2 can be reached after segment S_1 (e.g. wait(line6) to wait(line12) in Figure 4b).

Figure 4b shows an example Segment Graph where the only segment boundary is the function call of **wait()** (aka. Segment Graph[wait]). The corresponding input code for this graph is listed in Figure 4a. Note that the expression **z=z*z;** is part of both segments wait(line2) and wait(line6) because it can be reached from both **wait()** boundaries.

Note that our Segment Graph Generator differs in key aspects from the work in [7]. First, the user can parameterize the segment graph with any chosen segment boundaries (in [7] boundaries are hard coded). For instance, if the user selects all control flow statements as segment boundaries, the output will be a classic control flow graph (CFG). Alternatively, if we choose scheduler entry points (i.e. **wait**) as the segment boundaries, we obtain a Segment Graph for analysis of Discrete Event Simulation (DES) semantics. Second, our RISC compiler can handle recursive function calls which was not possible before. Third, we can process jump statements **break** and **continue**, as well as multiple **return** statements from functions. Before, valid input code could not include these statements. Also, expressions which have an undefined evaluation order will be properly ordered into a fixed sequence to avoid ambiguity. For instance, the expression **x=a+f()+b;** does not specify if the read access to the variables or the function call executes first. The RISC compiler translates this into **int t=f(); x=a+t+b;** where the function call always happens before the variable read.

3.2 Segment Graph Generator

The algorithm of our Segment Graph Generator is described in Listing 1 by the recursive function **build_graph**. The first parameter **curr_stmt** is the statement which will be processed next. The set **curr_segs** contains the segments which will consume the current statement. For instance, while processing the assignment **z=z*z** in Figure 4 the set **curr_segs** is {wait(line2), wait(line6)}, so the expression will be added to both segments. The **break** and **continue** statements represent an unconditional jump in a program. If we hit these keywords, the segments in **curr_segs** will be moved into the associated set **break_segs** or **continue_segs**, respectively. After completing the corresponding loop or switch statement, all segments in **break_segs** or **continue_segs** will be moved back to the **curr_segs** set.

For simplicity, we illustrate the processing of function calls and loops in Figure 5 and Figure 6, respectively. Figure 5 shows how the **build_graph** algorithm handles function calls. In step 1 the dotted circle represents the segment set **cur_segs**. The algorithm will detect the function call and check if the function is already analyzed. If not and it is the first time, the function will be analyzed separately, as shown in step 2. Otherwise, the algorithm reuses the cached graph for the particular function. Finally, each expression of segment 1 will be joined into each individual segment in the segment set 0. Segment 4 and 5 represent the new set **cur_segs**.

Correspondingly, Figure 6 shows the loop analysis for a while loop. Again in step 1 the dotted circle represents the segment set **cur_segs**. The algorithm detects the **while** statement and analyzes the loop body separately. The graph for the body of the loop is shown in step 2. Afterwards, each

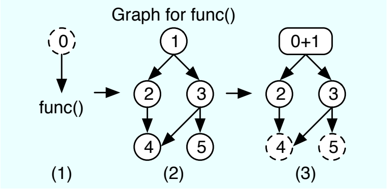


Figure 5: Function call processing

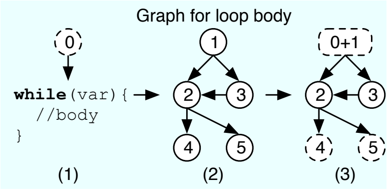


Figure 6: Loop processing

expression of segment 1 will be joined into the segment set 0. The new set `cur_segs` is the joined set of 0+1, 4, and 5. Note that we have to consider set 0+1 for the case that the loop is not taken.

```
SegSet build_graph(curr_stmt, curr_segs,
  break_segs, SegSet continue_segs) {

  if(isBoundary(curr_stmt))
    Segment new_seg; SegSet result(new_seg)
    foreach(seg in curr_segs)
      add_edge(seg, new_seg)
    return result

  if(isBasicBlock(curr_stmt))
    foreach(stmt in curr_stmt)
      cur_segs = build_graph(stmt, curr_segs,
                     break_segs, continue_segs)
    return cur_segs

  if(isIfStmt(curr_stmt))
    add_expression(if_condition, curr_segs)
    new_seg_set_1
      = build_graph(if_body, curr_segs,
            break_segs, continue_segs)
    new_seg_set_2
      = build_graph(else_body, curr_segs,
            break_segs, continue_segs)
    return join(new_seg_set_1, new_seg_set_2)

  if(isBreakStmt(curr_stmt))
    break_segs = join(break_segs, current_segs)
    curr_segs.clear
    return curr_segs

  if(isContinueStmt(curr_stmt))
    continue_segs = join(continue_segs,
                        current_segs)
    curr_segs.clear
    return curr_segs

  if(isExpression(curr_stmt))
    if(isFunctionCall(curr_stmt))
      // see Figure 5
    else
      add_expression(stmt, curr_segs)
      return curr_segs
}
```

Listing 1: Algorithm of Segment Graph Generator

3.3 Thread Communication Graph

Based on the generated Segment Graph, we then extract the *TCG* to aid designers who face legacy code that needs to be reused and revised. For this, we identify and pair the synchronization and communication points in the individual scheduling steps in the design.

The Segment Graph already determines which code elements are potentially executed in any given scheduling step. However, we have to add the edges for the identified synchronization and communication points. Specifically for event notifications, we have to analyze the `notify()` and `wait()` function calls in each segment. Additionally, we need to identify any channel communication calls, e.g `read()` and `write()`. Finally, the mapped channels and events are followed and matched through the design hierarchy.

3.3.1 Port Mapping

SystemC ports provide a flexible interface to send and receive data via mapped channels (or other mapped ports). While the indirect function calls via ports to channel methods are a powerful modeling feature, it is difficult to follow the actual flow of control in unfamiliar code. Here, our RISC compiler can help and determine which port is mapped to which channel, including for cases where this mapping goes through multiple levels of the design hierarchy.

To resolve port mappings, two steps are required. First, a port needs to be unambiguously identified in the hierarchy of the design. Second, we have to follow the module hierarchy to find the mapped channel.

For step 1, we identify a port in the design through a so-called *instance path*. An instance path is a list of tuples where each tuple contains a scope and an instance. For example, the path to a port DataIn could be [GlobalScope::top] → [Top::platform] → [Platform::datain] → [DataIn::port1]. Note that the instance path uniquely identifies a port, even if there are multiple instances of this port in the module hierarchy.

For step 2, we use the instance path to identify the mapped channel. Specifically, we analyze the mapping between a tuple and its successor. Here, we check the module constructor and identify the mapping to a channel or another port, and repeat the process as needed. In each iteration we go up in the instance tree until the port is mapped to a channel.

3.3.2 Event and Reference Mapping

SystemC threads use events for synchronization with each other. If the synchronizing threads are in the same module, a shared event variable in the module can be used. However, if threads have to synchronize across module boundaries, an event at a higher level in the hierarchy is needed, which is typically mapped via references.

Here, we can determine the reference mapping in the same fashion as the port mapping. We describe the event by an instance path and go up the path until the reference is mapped to an actual variable.

3.3.3 Handling of Loops

Loops are naturally present in virtually all algorithms and clearly need to be supported by our source code analysis. However, general loops can also lead to complex control flows that are difficult to represent cleanly in visual graphs. For the purpose of our TCG, we aim at a loop abstraction that simplifies the understanding of the protocol between the communicating parties. Specifically, our goal is to visualize the overall sequential flow of exchanged messages, in similar manner as exhibited by message sequence charts [10].

For our TCG, we support loops in one of two ways. On the one hand, we can assume that each loop will be taken at least once. On the other hand, loops can be unrolled. By default, our TCG assumes the first option because not every loop can be unrolled. Additionally, loop unrolling can lead to state explosion and often decreases the readability of the TCG.

In context of communication protocols, it is reasonable to assume that the sender and the receiver are exchanging messages equally. Listing 2 shows an example where `thread1` notifies `thread2` ten times. Here, the sender and the receiver are exchanging messages and thus the associated `notify()` and `wait()` functions must be called equally often. Figure 7 shows the TCG for the example in Listing 2. The graphs shows that the loop is taken at least once.

```
void thread1() {
  for(int i = 0; i < 10; i++) {
    event.notify(SC_ZERO_TIME);
    /* Do some stuff */
    wait(SC_ZERO_TIME); } }
void thread2() {
  for(int i = 0; i < 10; i++) {
    wait(event);
    /* Do some stuff */ } }
```

Listing 2: Producer and consumer example

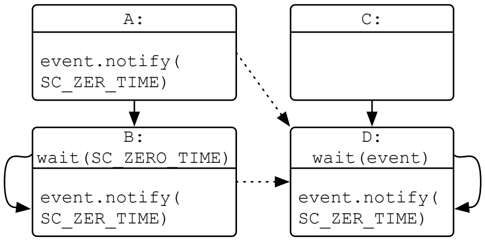


Figure 7: Example of a loop with synchronization

3.4 Module Hierarchy

We can use the SystemC IR to determine the module hierarchy of the design model in two steps. First, we identify the top module of the design. Unless explicitly specified by the designer, we assume that the top module is declared in the function `sc_main`. From the declaration, we can then derive the module definition. Finally, we can traverse the hierarchy of the design as described in Listing 3.

```
traverse_hierarchy(ModuleDefintion md) {
  foreach(ModuleInstance mi
      in get_all_sub_mdules(md)) {
    traverse_hierarchy(get_module_definition(mi))
  }
}
```

Listing 3: Traversing the module hierarchy

The function `traverse_hierarchy()` takes the module definition *md* and iterates over all its child modules. For each child module instance *mi*, the corresponding module definition is determined and the function `traverse_hierarchy()` is called recursively.

3.5 Optimization and Designer Interaction

The system designer has a set of configuration options and parameters available to get a better picture of the design. For example, the designer can select what types of edges should be displayed for the communication. Our TCG distinguishes native event notifications, primitive, and hierarchical channel communication.

The system designer can also select only a subtree of the module hierarchy or a subset of the SystemC threads for which the TCG will be generated. Thus, the designer can easily choose and focus on the points of interest and see the communication and dependencies between them.

Finally, pseudo comments may be used to indicate loop unrolling and `wait` statements can be annotated and labeled. This information is then displayed in the generated graph for enhanced readability.

Overall, the system designer can quickly and iteratively generate custom charts, getting more familiar, and obtaining a better picture of the model and its components.

3.6 Visualization

Our RISC compiler performs the analysis and graph generation based on the internal representation of the model and generates DOT files [11] for the Segment Graph and TCG. These files can then be visualized by the DOT tools, e.g. as interactive chart on screen or as PDF.

3.7 Accuracy and Limitations

The current implementation of our fully automated compiler has some limitations. We cannot handle pointers. Also, currently we cannot match array indices in port mappings.

Our compiler produces charts which are giving an impression of the communication behavior. However, static analysis can misinterpret situations and illustrate too many or too few communication edges. As mentioned, a picture is worth a thousand words (and even if a few words are inaccurate, the picture helps a lot in quick comprehension). For instance, in Figure 11 we can see immediately that the master $m1$ requests the bus through the arbiter. After the arbiter acknowledges the request, master $m1$ starts communicating to the slave. We should emphasize that our TCG generator is fully automated and quickly visualizes identified communication patterns without designer interaction. The generated illustrations may be inaccurate in minor aspects (limitations listed above), but they nevertheless convey an overall image that is helpful for getting to understand the source code quickly.

4. EXPERIMENTS

We have evaluated our TCG generation from SystemC source code on a Mandelbrot graphics application, an AMBA bus model, and the S2CBench benchmark set. For all examples, we tested both our hierarchical and communication analysis.

4.1 Mandelbrot Renderer

The Mandelbrot example computes a stream of Mandelbrot [12] images and as such is a representative for highly parallel graphics applications. The source code is complex, heavily instrumented with macros for customization. Here, we choose 2 parallel renderer modules.

The Mandelbrot module hierarchy is shown in Figure 8.

```
Top top
    +- DataChannel c1
    +- DataChannel c2
    +- Stimulus stimulus
    +- Platform platform
        +- DataIn din
        +- DUT dut
            +- main (thread)
            +- mb1 (thread)
            +- mb2 (thread)
            +- mb3 (thread)
            +- mb4 (thread)
        +- DataOut dout
    +- Monitor monitor
```

Figure 8: Module hierarchy of the Mandelbrot

We can deduce that the modules *Stimulus* and *Monitor* feed data in and out of the *Platform* and in turn the *DUT* (Design Under Test) (via *DataIn* and *DataOut*). The *DUT* hosts one main and four worker threads *mb1*, *mb2*, *mb3*, and *mb4*.

Next, we performed behavioral analysis and generated TCG, such as Figure 9. The *INIT* segments start threads and the solid arrows illustrate the transitions between the segments. The dashed arrows show the event synchronization. The generated Figure 9 focuses on the synchronization between the main and worker threads in the module *DUT*. We can see that the thread *main* notifies the worker threads *mb1*, *mb2*, *mb3*, and *mb4* when data is available and both respond back to *main*.

Figure 10 shows a higher level of communication analysis, where our RISC compiler first analyzed the module hierarchy and channel binding to identify the port mapping. Then it followed the port mapping through the different module levels and associated them with the corresponding `read()` and `write()` function calls. The resulting communication and data flow is generated in Figure 10. We can easily see that the data flows from *Stimulus* via *DataIn* into the *DUT* where the coordinates are processed. An image is then sent via *DataOut* to the *Monitor* module.

For both diagrams the assumption that loops are taken once fits very well. The combination of the structural and behavioral analysis provides quick insight into the behavior of the design.

4.2 AMBA Bus Model

As an example with complex multi-component communication, we reimplemented an AMBA bus model from [13] at TLM abstraction. The generated module hierarchy is shown in Figure 12.

The corresponding TCG for a BWRITE operation is shown in Figure 11. Here, the red dashed lines represent event notifications and blue dashed lines communication via channels. We can clearly see that master `M1` requests the bus through the arbiter via the event `areq1`. In turn, the arbiter grants the bus to `M1` via the `agnt0` event. Next, `M1` uses the bus to send data via `BD` to the `Slave`, which receives the `address` via the `Decoder`. Without this chart, the designer would need to read and study the 498 lines of code and manually figure out the port mappings and execution dependencies.

4.3 S2C Testbench

Finally we have evaluated our TCG generation on the S2CBench [14], a benchmark suite of 16 synthesizable SystemC models that includes industrial, automotive, security, telecommunication, and consumer applications. Our RISC compiler accurately generates the module hierarchy and TCG for these examples[1]. Due to space limitations, we unfortunately cannot present the resulting graphs here.

Each TCG was generated in less than 34 seconds, where on average the compiler spent 15 seconds on AST creation (ROSE parser), 10 seconds for SystemC IR generation, and 9 seconds on the TCG.

```
Top 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_to_bus _to_slave
    +- Channel decoder_to_slave
    +- Channel master1_to_slave _bwrite
    +- Master1 m1
    +- Master2 m2
    +- Arbiter arbiter
    +- DataBus databus
    +- Slave slave
    +- Decoder decoder
```

Figure 12: Module hierarchy of the AMBA bus

5. CONCLUSION

Becoming familiar with an unknown SystemC design is an often necessary and complex process. Designers have to identify communication patterns between threads and the behavior of individual components which can consume weeks of unproductive work.

In this paper, we propose the RISC compiler framework to analyze and identify structural, behavioral and communication aspects of SystemC models. Specifically, we automatically extract execution, synchronization and communication dependencies and visualize them quickly as Thread Communication Graphs (TCG).

The experimental evaluation of our framework using more than a dozen SystemC examples from different application domains shows that the automatically generated module hierarchy and visual communication charts are very helpful for system designers in becoming familiar with legacy or third party source code.

In future work, we plan to extend our approach for models with detailed timing and advanced TLM 2.0 semantics.

ACKNOWLEDGMENT

This work has been supported in part by substantial funding from Intel Corporation for the project titled "Out-of-Order Parallel Simulation of SystemC Virtual Platforms on

[1] For six of the 16 benchmarks, we manually replaced indexed array variables with regular ones, since our RISC compiler currently cannot match array indices in port mappings.

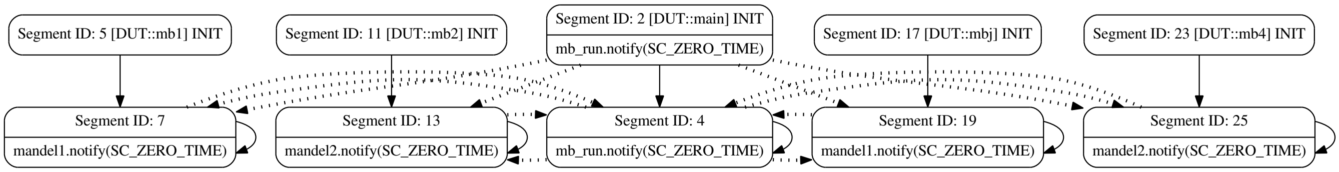

Segment ID: 5 [DUT::mb1] INIT
Segment ID: 11 [DUT::mb2] INIT
Segment ID: 2 [DUT::main] INIT
mb_run.notify(SC_ZERO_TIME)
Segment ID: 17 [DUT::mbj] INIT
Segment ID: 23 [DUT::mb4] INIT
Segment ID: 7
mandel1.notify(SC_ZERO_TIME)
Segment ID: 13
mandel2.notify(SC_ZERO_TIME)
Segment ID: 4
mb_run.notify(SC_ZERO_TIME)
Segment ID: 19
mandel1.notify(SC_ZERO_TIME)
Segment ID: 25
mandel2.notify(SC_ZERO_TIME)

Figure 9: Mandelbrot TCG for DUT

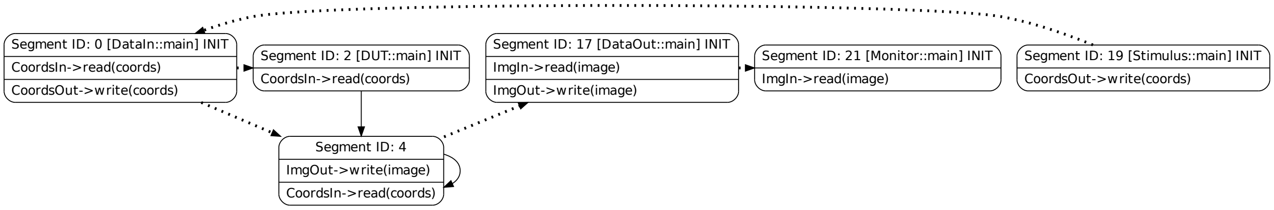


Figure 10: Generated Mandelbrot TCG with data flow from Stimulus via DUT to Monitor

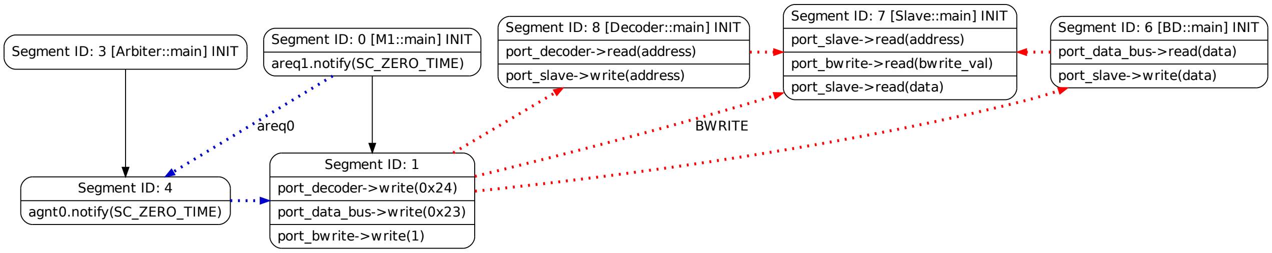

Segment ID: 3 [Arbiter::main] INIT
Segment ID: 0 [M1::main] INIT
areq1.notify(SC_ZERO_TIME)
Segment ID: 8 [Decoder::main] INIT
port_decoder->read(address)
port_slave->write(address)
Segment ID: 7 [Slave::main] INIT
port_slave->read(address)
port_bwrite->read(bwrite_val)
port_slave->read(data)
Segment ID: 6 [BD::main] INIT
port_data_bus->read(data)
port_slave->write(data)
areq0
BWRITE
Segment ID: 4
agnt0.notify(SC_ZERO_TIME)
Segment ID: 1
port_decoder->write(0x24)
port_data_bus->write(0x23)
port_bwrite->write(1)

Figure 11: Communication Graph generated from an AMBA bus model for a BWRITE operation

Many-Core Architectures". The authors thank Intel Corporation for the valuable support.

6. REFERENCES

[1] H. D. Patel, D. Mathaikutty, D. Berner, and S. K. Shukla, "CARH: Service-Oriented Architecture for Validating System-Level Designs," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 25, no. 8, pp. 1458–1474, 2006.

[2] "Doxygen," 2015. [Online]. Available: http://www.stack.nl/~dimitri/doxygen

[3] A. Kaushik and H. D. Patel, "Systemc-clang: An Open-source Framework for Analyzing Mixed-abstraction Systemc Models," in *Forum on specification and Design Languages (FDL)*, 2013.

[4] K. Marquet and M. Moy, "PinaVM: A SystemC Front-End Based on an Executable Intermediate Representation," in *Proceedings of the Tenth ACM International Conference on Embedded Software*, ser. EMSOFT, 2010, pp. 79–88.

[5] M. Moy, F. Maraninchi, and L. Maillet-Contoz, "Pinapa: An Extraction Tool for SystemC Descriptions of Systems-on-a-Chip," in *EMSOFT*, September 2005, pp. 317–324.

[6] N. Blanc, D. Kroening, and N. Sharygina, "Scoot: A Tool for the Analysis of SystemC Models," in *TACAS*, 2008, pp. 467–470.

[7] W. Chen, X. Han, and R. Dömer, "Out-of-Order Parallel Simulation for ESL Design," in *Proceedings of the Design, Automation and Test in Europe (DATE) Conference*, 2012.

[8] ——, "May-Happen-in-Parallel Analysis based on Segment Graphs for Safe ESL Models," in *Proceedings of the Design, Automation and Test in Europe (DATE) Conference*, 2014.

[9] D. J. Quinlan, "ROSE: Compiler Support for Object-Oriented Frameworks," *Parallel Processing Letters*, vol. 10, no. 2/3, pp. 215–226, 2000.

[10] S. Mauw and M. Reniers, "High-level message sequence charts," 1997.

[11] "Dot language," 2015. [Online]. Available: http://www.graphviz.org/doc/info/lang.html

[12] B. B. Mandelbrot, *Fractals and Chaos: The Mandelbrot Set and Beyond.* New York, Berlin, Paris: Springer, 2004.

[13] S. Roopak, R. Parthasarathi, and B. Samik, *Correct-by-Construction Approaches for SoC Design.* New York, Berlin, Paris: Springer, 2004.

[14] B. C. Schäfer and A. Mahapatra, "S2CBench: Synthesizable Systemc Benchmark Suite for High-Level Synthesis," *Embedded Systems Letters*, vol. 6, no. 3, pp. 53–56, 2014.