PREPRINT: "A. M. Kaushik and H. Patel, "systemc-clang: An Open-source Framework for Analyzing Mixed-abstraction SystemC Models," in proceedings of IEEE Forum on Specification and Design Languages (FDL), 2013, pp. 1-8."

systemc-clang: An Open-source Framework for Analyzing Mixed-abstraction SystemC Models

Anirudh Kaushik
Department of Electrical and Computer Engineering
University of Waterloo
amkaushi@uwaterloo.ca

Hiren D. Patel
Department of Electrical and Computer Engineering
University of Waterloo
hdpatel@uwaterloo.ca

***Abstract*—This work presents an open-source framework called systemc-clang for analyzing SystemC models that consist of a mixture of register-transfer level, and transaction-level components. The framework statically parses mixed-abstraction SystemC models, and represents them using an intermediate representation. This intermediate representation captures the structural information about the model, and certain behavioural semantics of the processes in the model. This representation can be used for multiple purposes such as static analysis of the model, code transformations, and optimizations. We describe with examples, the key details in implementing systemc-clang, and show an example of constructing a plugin that analyzes the intermediate representation to discover opportunities for parallel execution of SystemC processes. We also experimentally evaluate the capabilities of this framework with a subset of examples from the SystemC distribution including register-transfer, and transaction-level models.**

Keywords*—Design automation, Design methodology, Open source software.

I. Introduction

SystemC [1] provides a modelling and simulation framework for early design-space exploration of modern digital hardware systems. The SystemC standard allows for modelling systems at the register-transfer level (RTL) abstraction, and at abstraction layers higher than RTL known as transaction-level (TL). The transaction-level abstraction layer is particularly important during the early phases of the design cycle because it allows designers to model systems at a high-level of abstraction allowing them to focus primarily on the functional, communication and performance aspects of the model. Later in the design cycle the TL models, either in its entirety or only partially, are converted to RTL to allow for further detailed tradeoff and performance analysis for specific components in the model. Such models are known as mixed-abstraction models because portions of the model remain at TL and others at RTL. While SystemC provides a method to expedite the modelling and simulation of mixed-abstraction models, they pose a challenge to engineers responsible for building analysis and automation tools to assist in the design and verification process. This is because SystemC is a language built on top of C++, and analyzing SystemC models requires the capability to parse, represent, and comprehend the prescribed models. As a result, there is a need for frameworks that allow designers to analyze SystemC models in the electronic-design automation community.

In response to this need, several efforts propose frameworks to parse, and represent SystemC models [2], [3], [4], [5], [6], [7], [8]. These efforts fall into two broad categories: static approaches and dynamic approaches. Static approaches use static analysis methods to parse the SystemC source code, and extract structural and behavioural information that is stored in an intermediate representation. Examples that use static approaches include SystemPerl [2], KaSCPar [3], Scoot [4] and SystemCXML [7]. Notice that these approaches only support RTL models, and they do not support the analysis of TL models. An exception is HIFSuite [9], which is a commercial framework for analyzing both RTL and TL models.

Dynamic approaches, on the other hand, argue that certain aspects of the model may only be determined during run-time. Hence, it is important to determine these aspects while executing the SystemC models. One such framework is PinaVM [6]. An example where dynamic frameworks are useful include determining the model hierarchy where dynamic instantiation is used. While there is merit in this argument, the applicability of dynamic approaches for certain purposes such as model transformations, and synthesis is unlikely. Hence, we find that a concerted effort where parts of the analysis are done through static approaches, and others through dynamic approaches is the appropriate compromise. Consequently, we focus on developing an open-source framework that uses the static approach that we hope to integrate seamlessly with the existing dynamic approach proposed by PinaVM [6]. Note that PinaVM only supports the RTL abstraction layer whereas systemc-clang supports both RTL and TL. To the extent of our knowledge, only HIFSuite [9] supports the analysis of TL models. However, HIFSuite is a commercial tool, and we believe that an open-source framework for analyzing RTL and TL SystemC models is essential for the community. Hence, this work presents our efforts in building an extensible and open-source framework for analyzing mixed-abstraction SystemC models.

We call this framework as systemc-clang [10]. systemc-clang is built using clang[11], which is the front-end for the LLVM compiler framework. This is to allow for later integration with frameworks such as PinaVM. The main contributions of this work are as follows.

- We develop an extensible and open-source framework for analyzing SystemC models at both register-transfer level and transaction-level. Recall that existing open-source frameworks only support RTL, and do not

support TL. Furthermore, we capture the behavioural semantics of each process as a wait-state automaton [12].

- We provide a plugin interface for developers to seamlessly incorporate analyses, transformations, and code synthesis techniques with the framework.
- We illustrate the extensibility of systemc-clang by creating a plugin that implements an analysis to determine the possibility of parallel execution of processes in SystemC models [13].

II. Related Work

There are several prior efforts that build parsers for SystemC. Their purpose is similar to ours: to provide a framework for analyzing SystemC models. In particular, they were used for hardware synthesis, debugging, symbolic analysis, deadlock detection and translation. Some examples are SystemPerl [2], KaSCPar [3], and ParSysC [8]. They employ different techniques ranging from regular expressions to parse SystemC models to building or using existing C++ grammars with specific extensions to specify grammars for SystemC constructs [3], [8]. Other parsers such as Scoot [4], Pinapa [5] and PinaVM [6] use existing C/C++ front-ends such as GCC and LLVM. We broadly classify frameworks for analyzing SystemC models into static [2], [3], [4], [7], [8] and dynamic [5], [6] approaches. While static approaches are best suited for code generation, static type checking, and deadlock analysis, dynamic approaches extract details of the SystemC model that are only available at run-time.

SystemPerl [2] is a framework that uses Perl scripts to parse SystemC source files, and extract the structural information and hierarchy of SC_MODULEs. SystemCXML [7] is another framework that extracts structural information using Doxygen, and generates an intermediate representation of the SystemC model. However, both of these approaches are limited to extracting structural information, and they do not extract the behavioural semantics of the model. ParSysC [8] is a part of a larger framework called SyCE [14] that is not based on any existing C++ front-end. Instead, ParSysC uses the Purdue Compiler Construction Tool Set (PCCTS) [15]. Scoot [4] is a SystemC front-end based on GCC. In addition to extracting structural information, Scoot also supports type checking and code re-synthesis for faster simulation. PinaVM [6] is a more recent framework that draws inspiration from an earlier SystemC front-end called Pinapa [5]. Both works differ from the rest of the previous works in that they use a dynamic approach. This is done by parsing the bitcode of the model generated by LLVM's just-in-time compiler to elicit more information regarding the details of the model. The difference between the two is that Pinapa is based on the GCC compiler and only statically performs the analysis, and PinaVM is based on the LLVM compiler. This is important because LLVM continues to gain considerable prominence in research and open-source communities due to its community support, and research uses. Hence, we find that a framework that builds upon LLVM is likely to be well-supported. Furthermore, we observe that the above frameworks support only RTL models; thus, the key difference between our work and prior works is the support for TL models in addition to RTL models. We acknowledge that the commercial framework by HIFSuite [9], which is now part of EDALabs, supports TL models. However, to our knowledge, we believe that systemc-clang is the first open-source framework to support parsing models described at the TL, and the RTL. For a more detailed analysis of existing frameworks, their advantages and limitations, we direct the reader to read [16].

III. clang Background

We implement systemc-clang as a plugin for clang. clang is an open-source front-end for LLVM [17], and it is a compiler with active community and industry support. For example, industries such as NVIDIA, Intel, and AMD are all providing extensions to LLVM for specific language support and optimizations. The reasons for choosing clang over GCC's front-end is that clang provides expressive diagnostics and error reporting. It also provides access to the abstract-syntax tree (AST) that is user-friendly, and understandable. Furthermore, the clang community has a set of useful plugins such as Polly [18], a polyhedral optimizer for LLVM and the clang static analyzer [11], a source code analysis framework to automatically discover bugs. Its fast growing popularity, and its position as an alternative to GCC further motivates our choice to base systemc-clang on clang.

```
class FindPayloadCharacteristics: public
    RecursiveASTVisitor<FindPayloadCharacteristics> {
  public:
    /* Typedefs declarations */
    FindPayloadCharacteristics(CXXMethodDecl*, llvm::
        raw_ostream&);
    virtual bool VisitCXXMemberCallExpr(CXXMemberCallExpr
        *ce);
    /* Public member declarations */
  private:
  /* Private member declarations */
};

FindPayloadCharacteristics::FindPayloadCharacteristics(
    CXXMethodDecl* d, llvm::raw_ostream &os)
:_os(os)
, _d(d)
{
  TraverseDecl(_d);
}
bool FindPayloadCharacteristics::VisitCXXMemberCallExpr(
    CXXMemberCallExpr *ce) {
  if(ce->getMethodDecl()->getNameAsString() == "
      set_command") {
    FindArgument fa(ce->getArg(0)->IgnoreImpCasts());
    _command = fa.getArgumentName();
  }
  .......
}
```

Fig. 1: Code snippet of FindPayloadCharacteristics class.

clang provides two primary AST walkers: ASTVisitor and RecursiveASTVisitor. The key difference between the two is that RecursiveASTVisitor recursively visits all AST nodes whereas ASTVisitor does not recurse. In systemc-clang, we use the RecursiveASTVisitor that does a depth order traversal of the AST and visits each node. It performs three distinct tasks: traversing the AST using methods such as TraverseDecl(Decl*), TraverseStmt(Stmt*) and TraverseType(Type*); traversing the class hierarchy until the base class is reached and visiting a specific node using virtual overridable functions. For example, virtual bool VisitCXXMemberCallExpr(CXXMemberCallExpr*) is a virtual method that traverses nodes of type CXXMemberCallExpr.

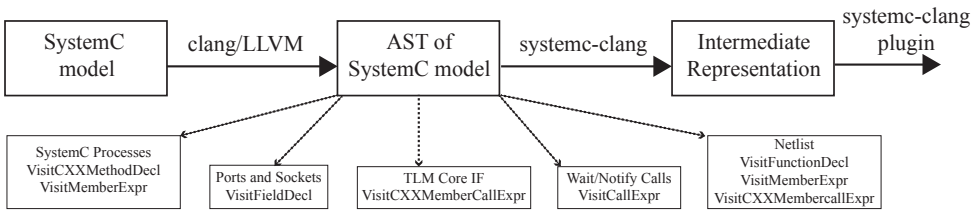


Fig. 2: Extraction of structural information and module hierarchy from AST of a SystemC model.

clang constructs	Description
Decl	Represents a declaration. Examples: CXXRecordDecl, CXXMethodDecl
Stmt	Represents a statement. Examples: WhileStmt, IfStmt, Expr, CallExpr
Type	Represents type hierarchy. Examples: VectorType, PointerType
TraverseDecl(...)	Traverses a declaration.
TraverseStmt(...)	Traverses a statement.
TraverseType(...)	Traverses a type.
VisitMemberExpr(...)	Visits AST node of type MemberExpr.
VisitCallExpr(...)	Visits AST node of type CallExpr.
VisitCXXMethodDecl(...)	Visits AST node of type CXXMethodDecl.

TABLE I: Fundamental clang classes, and methods.

The key base classes in clang are Stmt, Decl and Type. Table I lists the basic clang types and examples of classes inheriting from the basic classes.

Figure 1 shows a snippet of a class in systemc-clang that extracts the generic payload information communicated between initiators and targets via socket interfaces. A generic payload consists of a set of attributes that are set using certain public methods. For instance, the read/write command can be set using the set_command() access function and the starting address of the target memory map can be set using the set_address() method. Figure 3 shows the AST for setting attributes of the generic payload. It can be observed that the AST nodes for the methods set_command() and set_address() are of type CXXMemberCallExpr. Hence, in the FindPayloadCharacteristics class, we use the virtual overridable function virtual bool VisitCXXMemberCallExpr() to traverse nodes of type CXXMemberCallExpr. The public methods used to define the attributes of the generic payload take in a single argument. Therefore, we use getArg(), a public member function of the CXXMemberCallExpr class to extract the argument name for the particular attribute. Since the attributes for the payload are set in a SystemC process, the FindPayloadCharacteristics class constructor takes as argument an instance of CXXMethodDecl, which represents a method instance of a class/structure.

IV. SYSTEMC-CLANG TOOL FLOW

Figure 2 describes the tool flow for systemc-clang. We start by extracting the AST of the SystemC model using clang. This AST serves as an input to systemc-clang, which uses AST walkers to traverse the AST to extract RTL and TL specific information from the SystemC model. In Figure 2, we highlight certain information and the virtual methods used to traverse the AST node representing the information.

```
CXXMemberCallExpr 0x7fc160387928 'void'
|-MemberExpr 0x7fc1603878d0 '<bound member function type>
   ' ->set_command
0x7fc1606a0cb0
| `-ImplicitCastExpr 0x7fc1603878b8 'tlm::
   tlm_generic_payload *'
<LValueToRValue>
|   `-DeclRefExpr 0x7fc160387890 'tlm::
   tlm_generic_payload *' lvalue Var
0x7fc160386420 'trans' 'tlm::tlm_generic_payload *'
`-ImplicitCastExpr 0x7fc160387958 'tlm::tlm_command':'
   enum tlm::tlm_command'
<LValueToRValue>
  `-DeclRefExpr 0x7fc160387900 'tlm::tlm_command':'enum
     tlm::tlm_command' lvalue
Var 0x7fc1603869b0 'cmd' 'tlm::tlm_command':'enum tlm::
   tlm_command'


CXXMemberCallExpr 0x7fc160387a08 'void'
|-MemberExpr 0x7fc1603879b0 '<bound member function type>
   ' ->set_address
0x7fc1606a0f70
| `-ImplicitCastExpr 0x7fc160387998 'tlm::
   tlm_generic_payload *'
<LValueToRValue>
|   `-DeclRefExpr 0x7fc160387970 'tlm::
   tlm_generic_payload *' lvalue Var
0x7fc160386420 'trans' 'tlm::tlm_generic_payload *'
`-ImplicitCastExpr 0x7fc160387a50 'sc_dt::uint64':'
   unsigned long long'
<IntegralCast>
  `-ImplicitCastExpr 0x7fc160387a38 'int' <LValueToRValue
     >
    `-DeclRefExpr 0x7fc1603879e0 'int' lvalue Var 0
       x7fc1603867b0 'adr' 'int'
```

Fig. 3: AST for setting payload characteristics.

For instance, to determine the connections between ports we locate the bind method call or operator() by using CXXMemberCallExpr and MemberExpr to traverse the particular node and extract the connections. The information extracted from the parsed SystemC model is represented as an intermediate representation at three different levels: global, module and process level. systemc-clang plugins can obtain a reference to the entire intermediate representation of the model and perform further analysis. Although we support parsing of SystemC models at the RT level, we restrict the examples used to TLM 2.0 constructs as this is a key contribution of this work.

1) Extraction of structural information: The extraction of structural information begins with the AST of the input SystemC model. From the AST, systemc-clang first extracts the declarations of SC_MODULEs. Since SC_MODULEs are classes or structures, we use VisitCXXRecordDecl, the clang class that represents a structure or class definition. From the AST of each SC_MODULE definition, structural information regarding ports, signals, sockets, events, class data members and core interfaces are extracted. Port properties such as port access types (input/output/inout), and the data type (SystemC data types, native C/C++ data types or user

defined) are determined by traversing nodes of type FieldDecl. Sockets are specialized ports that allow communications in both directions between initiators and targets. The template arguments for the socket class define the width of data transferred through the socket and the base protocol policy used by the socket. The most commonly used socket classes are tlm_initiator_socket, tlm_target_socket, and the convenience sockets derived from these two classes. Standard sockets of type tlm_initiator_socket and tlm_target_socket require to be bound to objects that implement the corresponding transport interface. Convenience sockets on the other hand facilitate ease of implementing sockets in component design. They provide methods to register callbacks that relieve the need to bind sockets to objects implementing the transport interface; and convert between transport interfaces. simple_initiator/target_socket and passthrough_initiator/target_socket are examples of convenience sockets. However, unlike standard sockets, convenience sockets do not support hierarchical binding. Sockets are extracted in the same way as ports by traversing AST nodes of type FieldDecl. To determine the processes constituting the SystemC module, systemc-clang traverses the constructors of SC_MODULEs and locates AST nodes of type MemberExpr to elicit the process type, which can be of type SC_THREAD, SC_METHOD and SC_CTHREAD.

Information Type	Properties
Processes	Process Type, sensitivity, properties such as payload information and wait calls
Ports	Port Name, Port Access Type, Data Type
Events	Event Name
Class Members	Member Name, Member Type
Sockets	Socket Name, Protocol Type, Data bus Width, Socket Type, Callback Methods
wait Calls	Type of wait , Duration of wait , Event Name
notify Calls	Type of notify, Duration of notify, Event Name
Local Variables	Variable Name, Variable Type
TLM Payload	Payload Pointer, Command, Address, Data Pointer, Data Length, Streaming Width, Byte Enable Pointer, DMI Allowed, Initial Response Status, Extensions
TLM Core Interface	Socket Name, Interface Type, Payload Pointer, Delay Argument, Phase Argument Starting address, Ending Address

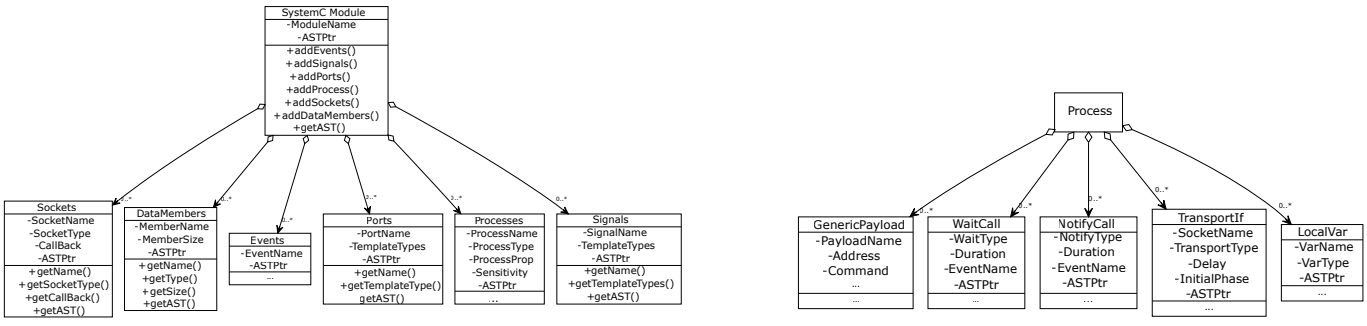
TABLE II: Information extracted from a given SystemC model.

Once the SystemC processes are extracted, systemc-clang iterates over processes and extracts information regarding wait and notify calls, local variables, transport interface details, and payload information used in communication between initiator and target for models described at the TL. For wait and notify calls systemc-clang determines the type of call based on the number of arguments and argument type. The AST node for wait calls and notify calls is the CallExpr class that has public member functions such as getNumArgs() and getArg() to extract the number of arguments and the argument type respectively. For instance, a single argument of the form SC_ZERO_TIME implies a delta wait or notification type. Similarly a notify call with no arguments represents an immediate notification. Local variables declared in processes are catalogued as variable name and data type. Table II summarizes the information and the characteristics that systemc-clang extracts from a SystemC model.

As systemc-clang also parses SystemC models described at the TL, information regarding the payload, and the core interfaces adopted by the modules can be extracted and analyzed. The generic payload is a data structure, which stores common attributes of memory mapped bus protocols such as starting address of target memory map, command (read or write), and Direct Memory Interface (DMI) access. systemc-clang extracts the attributes of the payload by traversing the AST nodes of public methods of the generic payload class that set attributes of the payload such as set_address, set_data_pointer, and set_streaming_width. The TLM 2.0 standard also provides extensions to the generic payload class to model user specific attributes of the payload. systemc-clang identifies any extensions to the generic payload class by traversing the AST node of the public method set_extension of the generic payload class. Another aspect of the TLM 2.0 standard are the core interfaces that define the interaction between the initiator and target. TLM 2.0 offers three different type interfaces: transport interfaces, direct memory interfaces and debug transport interfaces. The transport interfaces are used to communicate transactions between the initiator, interconnect component and target. There are two transport interfaces that support two different coding styles. Blocking and Non-Blocking transport interfaces support the loosely and approximately time coding styles. The blocking transport denoted by b_transport takes as argument the payload reference and a timing delay to indicate the start time and end time of a transaction. A blocking transport interface supports the loosely timing coding style where only the start and end times of a transaction are of interest. The non-blocking transport interface denoted by nb_transport_bw and nb_tranport_fw, supports the approximately timed coding style; which is suitable to model the interactions between the initiator and target. The non-blocking and blocking interface differs in an additional phase argument which denotes the initial phase of transaction for the non-blocking interface. Direct memory interface is another TLM core interface through which the initiator can access the target's system memory without communicating through the interconnect. The final class of core interfaces is the Debug Transport Interface, which relaxes the temporal constraints of the transport interfaces for software debugging purposes.

After extracting the necessary structural information at the module and process level, systemc-clang inspects the sc_main() function to extract the module hierarchy and simulation time (if specified). The netlist class makes use of the port and socket information extracted to determine the port bindings and module hierarchy. Since our approach relies on traversing the AST of the entire SystemC model to extract relevant information, the time for parsing the model is proportional to the size of its AST. However, appropriate use of clang's precompiled headers can assist in reducing this. Since systemc-clang is a static SystemC front-end, it cannot extract the binding of modules which are dynamically instantiated. The complete architecture of such a model is obtained after the elaboration phase. Hence, systemc-clang extracts the hierarchy of the SystemC model that is available at compile-time only.

2) Extraction of behavioural semantics: systemc-clang extracts the behavioural semantics of the SystemC model via a wait-state automaton. The wait-state automaton captures the



(a) IR at the module level.

(b) IR at the process level.

Fig. 4: IR at the module and process level.

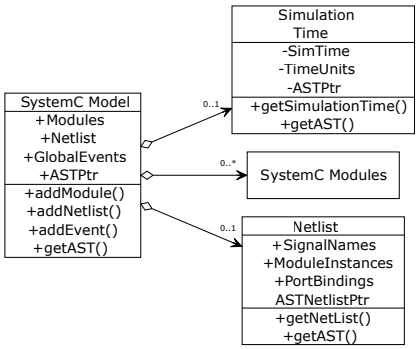


Fig. 5: IR at the global level.

semantics of the model, by representing the suspension points (wait calls) as states and code blocks between suspension points as state transitions. We borrow the structure of the wait-state automaton from Harrath et al. [12] and make slight modifications to account for event based suspensions for TL models.

A. Intermediate Representation

The intermediate representation (IR) is a collection of classes to store the parsed structural information with additional methods to query and display the AST for the particular information. We maintain this IR at three levels: global, module, and process level.

Figure 4a describes the IR structure at the module level. For ports and signals, template type field refers to the access type such as sc_in, sc_out, and sc_inout and data type of the port/signal, which includes SystemC data types, native C/C++ data types, and user defined data types. Figure 4b illustrates the information stored in the IR at the process level. Process information is stored as a mapping of the process name with its type, sensitivity list, AST node and other properties at the process level such as wait calls, notify calls, local variables, and payload information.

Figure 5 shows the class diagram of the IR maintained at the global level. At this level, information regarding the SystemC modules, netlist and simulation time (if specified) are recorded. In addition to the module name and AST pointer, the SC_MODULEs class also stores additional properties such as events, signals, ports, sockets, processes, and core interface. The key advantage of the IR is the pointer to the AST node for each information extracted. This allows any systemc-clang plugin to extract any further information from the AST node. This is beneficial as it allows further extensions based on the developers needs.

B. Extending systemc-clang with plugins

systemc-clang provides the feature of plugins that can be used to access the structural and behavioral information available to perform further analysis. This is done by inheriting the SystemCConsumer class, the main class responsible for parsing the SystemC model. The SystemCConsumer class provides a virtual method that can be overriden to obtain a pointer to the entire parsed SystemC model. One of the plugins that we make publicly available is ModelDump, which displays the parsed information in a graph structure where nodes represent SC_MODULEs and edges between nodes represent the port connections. The nodes contain additional information regarding the processes, data members and functions, wait calls, and notify calls while connections between nodes contain information regarding the port access types, socket transport type and payload information.

We list the basic steps in creating a systemc-clang plugin. We define a class ModelAnalysis that performs advanced analysis on the parsed SystemC model. It publicly inherits from the SystemCConsumer class that is responsible for parsing and maintaining the IR. The virtual method void postFire() is overridden to access the parsed SystemC model via the method call getSystemCModel() of the main class. With a reference to the parsed SystemC model, the developer can define classes/structs or functions to perform further analysis. We supply the class name ModelAnalysis as a template argument to the plugin class. On compilation, the executable will execute the ModelAnalysis plugin.

V. Experimental Evaluation

A. Example report for a SystemC TL Model.

We run systemc-clang on an approximately timed TL model with four initiators and four targets communicating through an interconnect. The source code for the example used is available at [19]. We show some of the information parsed by systemc-clang specific to TL modelling such as

```
For SC_MODULE : AT_interconnect
 Socket : init_socket
 Type : tlm_utils::multi_passthrough_initiator_socket
 Register Call Back Methods :
  register_b_transport register_nb_transport_fw
  register_get_direct_mem_ptr register_transport_dbg
  register_nb_transport_bw
      register_invalidate_direct_mem_ptr

 Socket : targ_socket
 Type : tlm_utils::multi_passthrough_target_socket
 Register Call Back Methods :
  register_b_transport register_nb_transport_fw
  register_get_direct_mem_ptr register_transport_dbg
  register_nb_transport_bw
      register_invalidate_direct_mem_ptr

For SC_MODULE : AT_typeA_initiator
 Socket : socket
 Type : tlm_utils::simple_initiator_socket
 Register Call Back Methods :
  register_nb_transport_bw


For SC_MODULE : AT_typeA_target
 Socket : socket
 Type : tlm_utils::simple_target_socket
 Register Call Back Methods :
  register_nb_transport_fw


For SC_MODULE : AT_typeB_initiator
 Socket : socket
 Type : tlm_utils::simple_initiator_socket
 Register Call Back Methods :
  register_nb_transport_bw
```

(a) Socket information.

```
For CXXMethodDecl : b_transport
  Socket : init_socket
  Interface details:
    Transport Type : b_transport Payload : trans Delay :
        delay

For CXXMethodDecl : nb_transport_fw
  Socket : init_socket
  Interface details:
    Transport Type : nb_transport_fw Payload : trans
        Delay : delay Phase : phase

For CXXMethodDecl : get_direct_mem_ptr
  Socket : init_socket
  Interface details:
    Transport Type : get_direct_mem_ptr Payload : trans
        DMI Pointer : dmi_data

For CXXMethodDecl : nb_transport_bw
 Socket : targ_socket
  Interface details:
    Transport Type : nb_transport_bw Payload : trans
        Delay : delay Phase : phase
```

(b) Core interfaces.

```
For CXXMethodDecl : thread_process in SC_MODULE:
    AT_typeA_initiator
 Payload : trans
 Command : cmd
 Address : adr
 Data Pointer : 16
 Data Length : 4
 Streaming Width : 4
 Byte Enable Pointer : 0
 DMI Allowed : false
 Response Status : TLM_INCOMPLETE_RESPONSE

For CXXMethodDecl : thread_process in SC_MODULE:
    AT_typeB_initiator
Payload : trans
Command : cmd
Address : adr
Data Pointer : 16
Data Length : 4
Streaming Width : 4
Byte Enable Pointer : 0
DMI Allowed : false
Response Status : TLM_INCOMPLETE_RESPONSE
```

(c) Payload information.

```
For SC_MODULE AT_TypeA_initiator:
  EntryFunctionContainer 'thread_process' processType '
      SC_THREAD'  CXXMethodDecl '0x7f2d5211d450
    WaitContainer numargs: 1  arglist: 'sc_core::sc_time(
        rand_ps(), SC_PS)'
    WaitContainer numargs: 2  arglist: '1' 'SC_NS'
    WaitContainer numargs: 1  arglist: 'this->
        end_req_event'

    Event Notification : this->end_req_event

For SC_MODULE AT_interconnect :
 Event Notification : this->end_rsp_event[ext->init]
 Event Notification : this->end_req_event[target]
 Event Notification : this->end_req_event[target]
 Event Notification : this->end_req_event[id]
 Event Notification : this->end_rsp_event[init]
```

(d) wait and notify calls.

Fig. 6: Examples of information extracted from a SystemC TL model.

socket information, core interface, payload information, wait calls and notify calls.

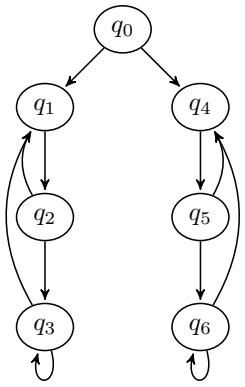
1) Socket Information: We extract the following information regarding sockets: name, type and callback methods for convenience sockets. Figure 6a illustrates the sockets used by the initiators, interconnect and targets. For instance, the sockets for the interconnect component are of type multi_passthrough_initiator_socket, which is a member of the convenience socket class. The initiator and target sockets are of type simple_initiator_socket and simple_target_socket respectively. All the sockets used in the example have a databus width of 32 and are based on the TLM base protocol.

2) Callback Methods: Convenience sockets can register callback methods thereby relieving the need for binding to an object that implements the necessary interface. Since, all the sockets used in the model are convenience sockets, systemc-clang locates callback methods registered for the socket. Figure 6a lists out the call back methods for all the sockets in the model. For instance, the initiator AT_typeA_initiator registers a callback method named nb_transport_bw for incoming non blocking transport from the interconnect component. The implementation of this transport interface is defined in the

```
void f1() {
  while(true) {
    wait(SC_ZERO_TIME);
    out.write(false);
    e.notify();
    wait(2, SC_NS);
    for (int i = 0; i<2; i++) {
      if(i == 0) {
        out.write(true);
      }
      else {
        out.write(false);
      }
      wait(4, SC_NS);
    }
  }
}
```

```
void f2() {
  int j;
  while(true) {
    wait(e);
    j = in.read();
    if(j < 5) {
      out.write(true);
      j++;
    }
    wait(1, SC_NS);
    for (int i = 0; i<2; i++) {
      j =i*8;
      wait(2, SC_NS);
    }
  }
}
```

(a) Simple SystemC example.



(b) Wait-State Automaton.

```
/******** Timing Analysis ********/
For Wait Call 0x7fcd1af13488 time tuple is 0 1
For Wait Call 0x7fcd1af13838 time tuple is 2 1
For Wait Call 0x7fcd1af13ec0 time tuple is 10 1
For Wait Call 0x7fcd1af15e38 time tuple is 0 2
For Wait Call 0x7fcd1af13c30 time tuple is 1 2
For Wait Call 0x7fcd1af16378 time tuple is 5 2
For Notify call 0x7fcd1af136a0 time tuple is 0 1

/******** Variable Access Analysis ********/
For wait call : 0x7fcd1af13838 Variable access types are:
 Port :out Write status : 1 Read status : 0
For wait call : 0x7fcd1af13c30 Variable access types are:
 Port   : in Write status : 0 Read status : 1
 Variable : j Write status : 1 Read status : 1
 Port : out Write status : 1 Read status : 0
For wait call : 0x7fcd1af13ec0 Variable access types are:
 Variable  :i Write status : 1 Read status : 1
 Port :out Write status : 1 Read status : 0
For wait call : 0x7fcd1af16378 Variable access types are:
 Variable  :i Write status : 1 Read status : 1
 Variable  :j Write status : 1 Read status : 0
```

(c) systemc-clang report.

Fig. 7: A timing analysis plugin for systemc-clang.

Case Studies	SystemC Ports	User Defined Ports	TLM Interfaces	SystemC Processes	Module Hierarchy	Behavioural Model
Canny Edge Detection	☑	N/A	N/A	☑	☑	☑
JPEG Decoder	☑	☑	N/A	☑	☑	☑
JPEG Encoder	☑	☑	☑	☑	☑	☑

TABLE III: Efficacy of systemc-clang.

initiator.

3) Payload Information: The payload is a data structure with a set of attributes commonly found in many memory mapped bus protocols. Some of the attributes are read/write command, address and streaming width. The generic payload is based on the default base TLM protocol with ignorable extensions to add specific attributes. Attributes for the generic payload are set using public methods of the generic payload class such as set_command(), set_address() and set_dmi_allowed(), which set the read/write command, specify the starting address of the target's system memory map, and provide a hint to the initiator to utilize the direct memory interface. Figure 6c illustrates the payload information for the generic payload communicated by one of the initiator classes. The OSCI TLM 2.0 standard [20] suggests three methods to extend the attributes of the generic payload and underlying protocol scheme. The first method makes use of the ignorable extensions of the generic payload using the base protocol. The second method suggests defining a new protocol type while the third method suggests defining a new protocol and transaction type. For payloads with ignorable extensions, we store a pointer to the class extension and display the class declaration. We are currently working on supporting parsing of user-defined protocol types and transaction payload and reserve it for future work. Figure 6b illustrates the payload pointer information, phase argument, delay argument, and memory address arguments extracted from the core interfaces implemented in the interconnect. For example, the transaction sent through socket init_socket is though a blocking transport interface and hence, only the payload pointer and the delay argument determining the start and end of the transaction is extracted. For transactions sent through a direct memory interface, the starting and ending address arguments are extracted as shown for socket targ_socket. Figure 6c illustrates the payload information set at the initiator modules. The attributes of the generic payload are extracted from the arguments to the public methods of the generic payload class such as set_address and set_data_pointer.

4) wait and notify Calls: Figure 6d shows the wait and notify calls across all for one of the initiator modules. systemc-clang extracts the event names and the duration for wait calls and stores them in a wait container class.

B. Efficacy of systemc-clang on image processing case studies

We run systemc-clang on three TL case studies: canny edge detection model, JPEG decoder, and a JPEG encoder to evaluate the information parsed by systemc-clang. We tabulate the information extracted in Table III. It can be observed that systemc-clang was able to extract enough detail to understand the module structure in terms of the module bindings, SystemC processes, standard and user-defined ports, and behavioral model in the form of wait-state automata for each SystemC process constituting the SystemC model. and behavioural model.

C. Application of systemc-clang plugins: Determining possible parallel execution in SystemC TL models

With recent research efforts focusing on accelerating SystemC simulations such as out-of-order execution [13] and utilizing multi-core CPUs and GPUs [21], we highlight a plugin for systemc-clang that implements the starting point of carrying parallel out-of-order simulation of system level design models proposed in [13]. While the authors of [13] implement the parallel out-of-order simulation for SpecC models, we use the base algorithms proposed in the paper to annotate

suspension statements with timing information and variable accesses (read/write) in code blocks between suspension points for each process. This information is embedded in the wait-state automaton of the process.

Figure 7a shows an illustrative SystemC code example with interspersed wait calls and notify calls. SystemC threads $f1$ and $f2$ belong to the same module and write to an output port of boolean type out. systemc-clang generates a wait-state automaton representation of the SystemC model from the control-flow graph. Figure 7b illustrates the wait-state automaton for the example code shown in Figure 7a. The initial entry state is denoted by q_0. q_1, q_2, and q_3 represent wait calls of thread process f1 and q_4, q_5 and q_6 represent the wait calls of thread process f2. Using the wait-state automaton, we calculate the time increments of the states and the read/write access of the variables and ports between suspension points. For instance, for the wait call wait(4, SC_NS) in process f1, the total simulation time increment caused by it is 10 SC_NS as it advances the simulation time by 8 SC_NS inside the loop and the additional increment in simulation time contributed by the wait call wait(2, SC_NS), results in a total of 10 SC_NS. Variable i has read and write status set to true between states q_2 and q_3 as it is incremented as well as read to check the loop condition. Port out has write status set to true as it is written between suspension points q_1 and q_2. Figure 7c shows the complete timing and access patterns for the example provided.

Based on the information generated, methods to detect hazards and scheduling conflicts can be implemented in this plugin to elicit opportunities for parallel out-of-order execution of SystemC models to realize faster simulation times. For instance, if the code between wait(SC_ZERO_TIME) and wait(2, SC_NS) in function $f1$ is executed in parallel with the code between wait(e) and wait(1, SC_NS) in function $f2$, the final value of the port out would be non-deterministic as there is a write-after-write (WAW) hazard. This is inferred by the read/write access patterns of port out corresponding to the $wait$ statements. However, based on Chen's methodology [13], at $f2$'s scheduling point in wait(1, SC_NS), we could execute $f1$'s segment after wait(2, SC_NS) out of order with $f2$ as there is no data dependencies and timing dependencies. The information provided in Figure 7c could be assimilated into look-up tables that could be used by the out-of-order scheduler to perform such scheduling.

VI. Conclusion

We present systemc-clang, an open-source SystemC framework for analyzing models at the register-transfer level and transaction level. We use the LLVM compiler to generate an AST of the model and traverse it using AST walkers. systemc-clang maintains an IR that stores information regarding the structure of the model and certain individual properties of the structure. We also provide a plugin interface, that can interact with the IR to perform further analysis on the parsed structure. With an example defined at the transaction-level, we illustrate the information collected by systemc-clang and highlight a potential use case of the plugin interface. Finally, we aim to continually support and enhance the tool for sophisticated structural analysis.

References

[1] Open SystemC Initiative, "SystemC," http://www.systemc.org.

[2] W.Snyder, "Systemc-Perl," http://www.veripool.org/wiki/systemperl.

[3] FZI Microelectronic System Design, "KaSC-Par - Karlsruhe SystemC Parser Suite," http://www.fzi.de/index.php/de/component/content/article/238-ispe-sim/4350-sim-tools-kascpar-examples.

[4] N. Blanc, D. Kroening, and N. Sharygina, "Scoot: A tool for the analysis of systemc models," in *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 467–470.

[5] M. Moy, F. Maraninchi, and L. Maillet-Contoz, "Pinapa: an extraction tool for systemc descriptions of systems-on-a-chip," in *Proceedings of the 5th ACM international conference on Embedded software*. ACM, 2005, pp. 317–324.

[6] 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*. ACM, 2010, pp. 79–88.

[7] H. D. Patel, D. A. Mathaikutty, D. Berner, and S. K. Shukla, "CARH: Service-oriented architecture for validating system-level designs," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 25, no. 8, pp. 1458–1474, 2006.

[8] G. Fey, D. Große, T. Cassens, C. Genz, T. Warode, and R. Drechsler, "ParSyC: an efficient systemc parser," in *In Workshop on Synthesis And System Integration of Mixed Information technologies (SASIMI*. Citeseer, 2004.

[9] B. Nicola, D. G. Giuseppe, F. Michele, F. Franco, P. Graziano, S. Francesco, and V. Alessandro, "HIFSuite: tools for hdl code conversion and manipulation," *EURASIP Journal on Embedded Systems*, vol. 2010, 2010.

[10] Hiren D. Patel and Anirudh Kaushik, "A SystemC Parser using clang," https://github.com/hdpatel/systemc-clang.

[11] clang Team, "clang: a C language family Frontend for LLVM," http://www.clang.llvm.org.

[12] N. Harrath, B. Monsuez, and J. Delacroix, "Building systemc waiting state automata," in *Proceedings of the Fifth international conference on Verification and Evaluation of Computer and Communication Systems*. British Computer Society, 2011, pp. 108–119.

[13] W. Chen and R. Dömer, "Optimized out-of-order parallel discrete event simulation using predictions," in *Proceedings of the Conference on Design, Automation and Test in Europe*. EDA Consortium, 2013, pp. 3–8.

[14] R. Drechsler, G. Fey, C. Genz, and D. Große, "SyCE: An integrated environment for system design in systemc," in *Rapid System Prototyping, 2005.(RSP 2005). The 16th IEEE International Workshop on*. IEEE, 2005, pp. 258–260.

[15] T. J. Parr, H. G. Dietz, and W. E. Cohen, "PCCTS reference manual: version 1.00," *ACM Sigplan Notices*, vol. 27, no. 2, pp. 88–165, 1992.

[16] K. Marquet, M. Moy, and B. Karkare, "A theoretical and experimental review of systemc front-ends, verimag research report," 2010.

[17] LLVM Team, "The LLVM Compiler Infrastructure," http://www.llvm.org.

[18] ——, "The LLVM Compiler Infrastructure," http://polly.llvm.org/.

[19] Aynsley, John, "Complete tlm-2.0 at example," http://www.doulos.com.

[20] "OSCI TLM 2.0 Standard," 2008, http://www.systemc.org.

[21] R. Sinha, A. Prakash, and H. D. Patel, "Parallel simulation of mixed-abstraction systemc models on GPUs and multicore CPUs," in *Design Automation Conference (ASP-DAC), 2012 17th Asia and South Pacific*. IEEE, 2012, pp. 455–460.