A

Parsing SystemC using C/C++ Front-end Compiler

This chapter\(^1\) presents a more involved, but robust approach for parsing SystemC for structural and behavioral information using the Edison C/C++ Front-end Group compiler (EDG) [28]. This is a commercial product that allows parsing C/C++ source code and we use it for parsing SystemC. We document our infrastructure in non-intrusively adding our parser (from here forth it will be called walker) with EDG and then detail on how we extract certain information from the source. Our walkers should work with version of EDG that are higher than version 3.8 given that certain crucial properties of their internal data structures have not changed. We also use Autotools [131] to simplify the deployment of our parser, called FEDG.

![Fig. A.1. Parsing Flow Using EDG](image)

A.1 Tool Flow

The parsing of SystemC source begins by specifying the SystemC source files as input to the tool (Figure A.1). The source files are then run through EDG’s C/C++ front-end compiler that creates an intermediate language (IL)

\(^1\) Derek O’Neill is a co-author of this chapter.
representation; a complex data structure in the form of a tree, that contains the C/C++ equivalent information about the source files. Since our needs require parsing SystemC constructs, we take the EDG IL tree as input to a set of walkers that consist of functions to traverse the EDG IL tree and populate secondary data structures. Our library, FEDG Library contains these walkers that are used to extract SystemC constructs. The secondary representation uses the Xerces Library to create an XML document object manager (DOM) that only stores the information relevant to SystemC.

### A.2 Parsing SystemC

EDG’s IL representation of C++ consists of several basic building blocks. Each source file, function definition, and class declaration is represented as a scope. Sections of source code that are grouped together by braces are generally represented as a scope object. Contained in a scope object is likely to be a block of statements. This ‘block’ is a sequential list of each individual line of code inside the scope. Each statement in this list has its own type, the most common of which is an expression (but it could also be an initialization, for example). Each expression consists of expression nodes, logically linked together by operands, if the expression represents an operation. Each of these structures will be discussed in greater detail when they are encountered later in this section.

Structural parsing begins with a call to the EDG function walk_file_scope_il. This function accepts a callback function as a parameter, and will pass that callback function a pointer to each entry in the file-scope IL tree. In the EDG IL tree, a scope is the term for all of the code contained within a set of braces. This means that each function body is represented as a scope, as well as the statements inside the braces of a while loop. There is an EDG object that represents this section. There is also a special scope, the primary scope, that is not defined by a set of braces, but rather represents the entire source. This scope contains all of the other scopes. The aptly named callbackFunction in moduleparsing.cpp processes each of these entries, looking for ones that represent SystemC modules. We are looking for an SC_MODULE macro definition. The expansion of this macro reveals a class definition specified by the argument of the macro and an inheritance relationship with the class sc_module. So, next we check if the scope is of the kind that indicates a class, struct or union declaration. We also know that a class that is an SC_MODULE declaration has a base class of sc_module, so we check the list of base classes to make sure we are looking at a definition that inherits from a sc_module definition. If we find that our class scope has a base class of sc_module, we begin extracting the structure of the sc_module (with a function called insertNewModule). This allows declarations of module classes without using the SC_MODULE macro. For example, the parser will recognize class memory: public sc_module as a module declaration as well.
Ports of the module are represented as data members of the module class. These data members in the IL tree, are contained in a linked list that is a member of the EDG structure associated with the class. By traversing that list and checking each element’s type for \texttt{sc\_port}, we can determine if any of the fields are in fact ports. We must also check the base classes of each field, though, because a port may have inherited from the class \texttt{sc\_port}, and not actually be an instance of it. We do this using the base classes list associated with the field. Since the data type of a port is given by a template argument (and sometimes by a type with a template argument of its own), we must get the template arguments using a function called \texttt{getTemplateArg}. While we are traversing the list of fields, we also check to see if any of the fields are derived from \texttt{sc\_module}. If we find one, we know that we have encountered an instance of a different \texttt{sc\_module}. With that instance we call \texttt{addModuleInstance} to further parse this field.

Next, we move on to identifying the processes defined by this \texttt{sc\_module}. Processes are defined in the constructor of the module either via the use of macros or the construction definition of the module class. We parse the source such that either methods of definition are correctly extracted. We start by traversing the list of routines that this class declares. Once we find the routine with the same name as the module itself (the constructor), we use the EDG function \texttt{walk\_routine\_scope\_il} to traverse it. We are looking for calls to \texttt{create\_*\_process}, where \(*\) is either method, thread, or cthread. Once we have found all of these calls, we know we have a complete list of the module’s processes. This also handles multiple declarations of processes.

After the processes have been identified, \texttt{identifySensitivity} is called to parse the sensitivity expressions associated with each process declaration. The \texttt{identifySensitivity} function verifies that we are dealing with a block of statements and then calls the \texttt{parseStatements} function to check each one for a call to the \texttt{sensitive} function. EDG’s IL tree represents any group of statements, such as those inside a while loop or even those inside the same function, as a block of statements. This block contains a linked list of statement objects, each one representing one line of code from the original source. Since a sensitivity list in SystemC is effectively a call to the sensitive function, it will be represented as one statement. Traversing a block of statements allows us to identify that sensitivity statement for parsing. Function \texttt{parseStatements} will recursively call itself when it finds a nested statement block, and it will call \texttt{parseExprForSensitivity} when it finds an expression identified by EDG. The function \texttt{parseExprForSensitivity} attempts to traverse the expression tree of a valid sensitivity expression of a SystemC process (which is expressed in the constructor). The operands pointer in an expression node points to a list of one or more operands that make up the expression. Some operations have two or more operands (such as the addends in an addition operation), while others may have only one (such as the variable to be negated by the unary negation operator). When parsing operations, it is usually necessary to traverse the expression tree using these operands pointers. The SystemC
implementation overloads the $<$ operator to specify the signals that are to be made sensitive. This amounts to identifying a function call for $\text{operator} < <$ in the IL tree. If a function call is found in the expression, and the routine being called is $\text{operator} < <$, it calls the function $\text{scan_sensitivity_expression}$. This function traverses the expression tree in all of the ways that would be valid for different sensitivity expressions.

When defining a hierarchical module, instances of other modules and their connections are specified in the constructor. To identify these connections between modules that are declared in this constructor, the $\text{findConnectionsInRoutine}$ function is invoked. This function iterates over the statement list of the constructor, checking each statement for the sequence of expression nodes that occurs with each different type of connection ($\text{sc_clock}$, $\text{sc_signal}$, $\text{sc_in}$, $\text{sc_export}$, etc.). Since there is a different sequence of next and operands pointers to follow for each different signal type in a connection, this routine must check each one to find out which type of connection it has encountered. A separate case must also be considered for port-to-port binding (when a signal is connected using the syntax: $(\text{in}(\text{gate} \rightarrow \text{out}))$, since there is yet another different sequence for this scenario.

All of the previous parsing was done inside the $\text{sc_module}$ definition. The rest of the structural parsing described here is done inside the $\text{sc_main}$ function. It includes no new methods of parsing, but instead it is just different ways of obtaining information about instances and connections that are declared in $\text{sc_main}$.

Instances can be declared in two ways: an object instance of the module and a dynamically created module instance. These instances in $\text{sc_main}$ are identified by the function $\text{identifyInstances}$. The first task this function does is to find the $\text{sc_main}$ routine. The function iterates over every routine in the file scope until it finds the routine called $\text{sc_main}$. Then, the $\text{identifyModuleInstances}$ function looks at every statement in $\text{sc_main}$ for one that is an initialization. If it is an instance of an $\text{sc_module}$, it will be a dynamic initialization expression or a constructor.

Connections in $\text{sc_main}$ are identified by the function $\text{identifyConnections}$. This function again identifies the $\text{sc_main}$ routine by iterating through the list of routines in the top-level object from which all IL objects can be accessed. Once it finds that scope it passes it to the $\text{findConnectionsInRoutine}$ function, which operates exactly as it did before (when it was searching the constructor of an $\text{sc_module}$) to find the connections. If a function called from within the constructor contains connection information, that information is currently not parsed. To incorporate this functionality, the $\text{findConnectionsInRoutine}$ method could be recursively passed the pointer to any functions called within the constructor.

Behavioral parsing begins in much the same way as structural parsing does, with a call to $\text{walk_file_scope}$.il with a callback function. The $\text{writefinder}$ callback function again looks for $\text{sc_module}$ definitions and calls
parseStatementsForWrite to identify the constructs we are looking for. The `parseStatementsForWrite` function checks for many common behavioral constructs that will be required to parse behavioral information from the module. It currently checks for if statements, loops, switch statements, and return statements. When it encounters one of these elements, it adds the appropriate elements to the XML document object manager (DOM) and recursively parses the statements inside the element. The `parseExpression` function is used to build a string representation of the main expression of the statement (the test condition for an if statement, for example). Since there are over 150 different possible expression types, the `parseExpression` function currently only works for a subset of expressions, such as equality operations, inequality operations, and function calls.

Along with checking for these behavioral constructs, the expressions of the loops and if statements are checked for calls to `wait`, `write` on ports and signals, and `read` on ports and signals by the function `traverseAnExpr`. It uses EDG's built-in `traverse_expr` function, which operates like `walk_file_scope_il` in that it is passed a callback function and it returns a pointer to each expression node in the tree to that function. In our callback function we examine each node in the expression and look for a routine call to either `write`, `read`, or `wait`. At this time, any function called `write` or `read` will be identified by this method. To be more correct, however, the algorithm should verify that the call is made on an `sc_signal` object (or a derivative of `sc_signal` in `sc_signal` if). This would prevent a false positive identification of a user-defined function of the same name. Once a `read` or `write` call is found on any object, the function follows the correct sequence of operands and next pointers to get the port name and the variable that is being accessed.

An XML DOM is used to store all the behavioral and structural information in an XML format built on top of [123]. The format of our DOM follows the .dtd file included with the distribution. An example of initialization of the DOM can be seen in the function `initXMLDocument` function in `Back_End.cpp`. The DOM is printed to a file using the `justserialize` function in `structuraltoxml.cpp`, adapted from the print example in the Xerces distribution. The `DOMDocument->createElement(X("name"))` call is used to create a new element in the XML document. The attributes of that element can be set using the `DOMElement->setAttribute(X("name"), X("value"))` call. Children of the element can be created as new independent elements and can be appended to the parent with the `DOMElement->appendChild(element)` call. After being run on an input file, the program will output the DOM to a file called xmloutput.xml in the root of the program's installation.
B

Eclipse-based Plugin for a SystemC IDE

B.1 Project Overview

Requirements: Eclipse 3.2.x [126]; CDT 3.1.x [124]; a C++ compiler
Platforms: Any platform that Eclipse and CDT support.
Installation: 1. Use the Eclipse Update Manager to use the following URL
as the Remote Site
   http://fermat.ece.vt.edu/tools/systemcide/scide_site/.
   2. Download the zip file, and extract the folders into the Eclipse root.

The SystemC IDE [127] project\(^1\) is organized into an Eclipse plug-in and feature. The feature governs all of the plug-ins that are related to the SystemC IDE. At this point, there is only one plug-in for the project, but in the future, as the project grows, it may be reasonable to split the project into multiple plug-ins. The feature also allows the project to be distributed through the Eclipse Updates Manager. Having a familiarity with Eclipse plug-in and feature projects will definitely provide an advantage for future modifications.

B.2 SystemC IDE Feature

The only file of concern in this feature is feature.xml. Eclipse will auto-generate a lot of the details of the feature through wizard inputs. Opening this file in Eclipse will open it in the Feature Editor. Under the Plug-ins tab, edu.vt.fermat.scide has been added as a project that it governs. If Eclipse has not done it, dependencies must be calculated by going to the Dependencies tab and clicking the Compute button. Dependencies are the other plug-ins (for this project, some are Eclipse-provided, others are CDT).

---

\(^1\) Nick Sandonato is a co-author of this chapter.
B.3 SystemC IDE Plug-in

The governing file for a plug-in is plugin.xml. Much like the feature, dependencies are defined in the Dependencies tab. These are not automatically calculated like with the feature project. Additionally, it is possible to view the XML source code for the plug-in by clicking on the plugin.xml tab. In this file, the SystemC IDE project extends other plug-ins to provide the environment’s functionality. Explaining everything in this file would be outside of the scope of an overview. Understanding it would require some reading of Eclipse plug-in development, and also understanding some of the CDT project extension points [126, 124] (Figure B.1).

B.3.1 Plug-in Source

The plug-in project has been broken up into multiple packages. Each package has a fairly self-explanatory naming convention. Classes of notable interest:

- SystemCEditor – Responsible for presenting a SystemC source editor (at this time, focusing on SystemC-specific syntax highlighting)
- SystemCCodeScanner – Responsible for adding SystemC keywords to the editor’s parsing rules and add specific coloring to these words
- SystemCPreferencePage – Responsible for providing user-stored preferences for the SystemC development environment
- NewManagedSystemCProjectWizard – Responsible for creating a managed SystemC project (i.e., the Makefile is generated for the user). Additionally, this class is responsible for making sure the include and library paths for a SystemC project are set up properly
B.4 Setting up the SystemC IDE

At the moment, SystemC macros and types are colored the same as C++ keywords. In the future, it is definitely possible to define a new page in SystemCPreferencesPage to regulate macro and type colors. This would also require drawing a difference between macros and types in SystemCKeywordSets.

B.4 Setting up the SystemC IDE

This assumes that one has already installed Eclipse 3.2, CDT 3.1, and SCIDE 1.0. Starting up Eclipse for the first time will present the user with an environment as in Figure B.2.

B.4.1 Setup the SystemC Preferences

The user must specify where the SystemC include and library paths are. To accomplish this, modify the SystemC Preferences (Figure B.3) by going to: Window > Preferences... > SystemC.

B.4.2 Creating a Managed SystemC Project

The easiest way to manage a SystemC project is to allow the environment to manage the Makefile and building of the project. To create a managed SystemC Project: File > New > Project > Managed Make SystemC Project.
Supply a project name then select Finish. The workspace will confirm a perspective change to the SystemC Perspective. Accept this perspective change, and there will be a new project in the workspace (Figure B.4).
B.4.3 Adding Files to the Project

There are various ways to add source (or any other) files to a SystemC Project. It is possible to use the SystemC wizards to create source files, the unique wizard is the Module Creation Wizard. This wizard will automatically generate a header file that contains a module definition. Alternatively, source files may be imported from other projects through Eclipse’s Import Wizard. The Import Wizard can be accessed by right clicking on the project and selecting Import? Navigate to the source files, and Finish the import process. Either of these methods will trigger the CDT’s automatic build process.

B.4.4 Executing a SystemC Project

To execute a SystemC project, the previous steps must be completed. If there are no errors with the project (Figure B.5), the project can be run.

Fig. B.5. The Eclipse Problems View Reporting a Warning

Select Run > Run.... Select C/C++ Local Application, and then click the New icon.

Next, specify the project and Search the project for the project’s executable (Figure B.6).

Finally, select Run to execute the program. The Console View (Figure B.7) will display the output of the program—if any.

B.5 A Little About Implementation

The SystemC Integrated Development Environment (SCIDE) is truly an example of the extensibility and reusability of the Eclipse and CDT environments. In particular, SystemC’s relationship with the C++ proves to be very advantageous in minimizing development time and code duplication.

B.5.1 The Plug-In File

The file plugin.xml plays an important role in Eclipse projects. It is at the heart of how the project will extend the Eclipse environment. The various extensions tell the Eclipse environment how to make use of the classes developed in the following sections. A full-out discussion of the file is outside of the scope of this paper, as there are many excellent resources available; however, there are no portions of this that stand out in anyway.
B.5.2 The Editor

A majority of the SystemC Editor’s functionality-like content assist is taken care of by the CDT’s CEditor class. To incorporate the SystemC language’s keywords into the editor for syntax highlighting purposes was made easier since the language is a superset of C++. The SCKeywords class, while adding SystemC’s keywords, extends the CDT’s Keywords class to provide the keywords of C++.

The SystemCEditor class initializes the editor by setting the source viewer configuration, which is a class that sets up a code scanner for the SystemC language. The code scanner, SystemCCodeScanner, utilizes the keywords defined by SCKeywords to create a rule set for how text should be highlighted. There are various Rule-type classes, to which individual rules, or words, are
B.5 A Little About Implementation

added. These rules are all collectively added to a generic list of rules that
the `createRules` method of the `SystemCCodeScanner` returns. The SystemC
Editor is now capable of applying these rules to provide syntax highlighting.

B.5.3 The Preference Page

The `SystemCPreferencePage` is currently the primary preference page for
SystemC Projects. This class extends the traditional Eclipse preference page,
and allows developers to set the SystemC library, compiler options, and in-
clude file locations. Currently, these preferences must be defined before a
SystemC Project is created in order for the project to be aware of these spec-
ified preferences. The Preference Page utilizes what is known as a Preference
Store, which is defined by the `SystemCPlugin` class. The Preference Store
will maintain the SystemC preferences that have been saved for the current
workspace. Changing workspaces, however, will result in a different set of
SystemC preferences.

The actual control is built in the method `createContents` where labels,
text areas, buttons, and listeners are initialized. Each button requires a listener
that defines what happens; generally, a dialog is opened for this preference
page. Additionally, there are a few buttons that are built into the page auto-
matically: Restore Defaults, Apply, Cancel, and OK. There are overrideable
methods (`performDefaults` and `performOk`) that handle the corresponding
actions. It is worthwhile to note that the Apply button will actually invoke
the method `performOk`. The method `performOk` will take the values in the
text areas of the control, and save them to the Preference Store that is de-

defined for this workspace. The method `performDefaults` will set the text areas
of the control to contain the default values defined by the SystemC plug-in.
 Currently, the preferences are very environment-specific, so they are empty
strings by default.

B.5.4 The Wizards

The SystemC IDE has numerous wizards to aid in development. Most of the
wizards are fairly self-explanatory, ranging from creating an entire SystemC
project down to the specific source files that make up the project. A ma-

ority of the wizards take advantage of the extensibility of the CDT plugin,
modifying window titles and images associated with the wizard.

The `NewManagedSystemCProjectWizard` is the workhorse of the SystemC
IDE. This class sets up a SystemC Project that, through underlying capabili-
ties of the CDT, maintains the Makefile needed to build the project. Because of
this automation, and SystemC’s dependency on external libraries, it was nec-

essary for the wizard to add some additional configurations. The preferences
that are saved in the SystemC Preference Store are added to the configura-
tion for the underlying CDT project through the overridden `doRun` method.
The method manipulates the build information of the new project using the CDT’s ManagedBuildManager.

The NewModuleCreationWizard is a SystemC-specific wizard. Much like the CDT has a Class Creation Wizard, SystemC has a Module Creation Wizard. At its current stage, the wizard will generate a header file containing a new SystemC Module with some of the basic elements of a module. The NewModuleCodeGenerator class, which extends the CDT’s NewClassCodeGenerator class, is responsible for generating the file and adding the template code.
References


References

80. P. Mishra, N. Dutt, N. Krishnamurthy, and M. Abadir. A Top-Down Methodology for Valida-
81. P. Mishra, N. Krishnamurthy, N. Dutt, and M. Abadir. A Property Checking Approach to Micro-
86. OMG. OMG. http://www.omg.org/.
87. OMG. OMG CORBA. http://www.corba.org/.


133. VHDL. VHDL. http://www.vhdl.org/.