Design Structure Analysis and Transaction Recording in SystemC Designs: A Minimal-Intrusive Approach

Wolfgang Klingauf, Manuel Geffken
TU Braunschweig, E.I.S. (Prof. U. Golze)
38106 Braunschweig, Germany

Abstract

We present an introspection/reflection framework for SystemC which extracts design-relevant structure information and transaction data under any LRM-2.1 compliant simulation kernel without the need for kernel modifications or a parser. The proposed methodology requires just minimal changes to the user's source code and provides an extensible interface for sending the collected information to a database or interacting with an analysis tool, e.g. via sockets. We specify an XML schema for managing the data gathered by our framework and show how Java-based analysis tools can easily be written with this approach.

1 Introduction

SystemC is gaining more and more in importance for modern System-on-Chip design flows. The new version 2.1 of the language, which recently has been successfully approved by the IEEE as a standard [18], brings major improvements in language consistency and usability, and therefore promotes SystemC as an ideal base for electronic-system-level design. As the language traditionally focuses on the communication-centric transaction-level modeling (TLM) paradigm [4], many extension frameworks, design patterns, use cases, and design flow proposals for transaction-level modelling with SystemC are already available. A comprehensive overview of the state-of-the-art is given in [12] and [16].

However, SystemC still is a young language and thus tool support is limited. In particular, even basic everyday programming utilities such as design structure visualization and model hierarchy browsing or transaction-aware communication analysis are not available in a universal and easy-to-use manner. There do exist some mature SystemC/TLM development frameworks with nice linter, debugging and visualization capabilites such as AccurateC [1], ConvergenSC [6], Celoxica's Agility SystemC/RTL compiler [5], or Summit's SystemC Design and Verification Platform Vista [22]. But all of these tools are either limited to a certain SystemC subset [5] or application domain [6], and/or are bound to a specific version of the SystemC language, often in combination with a modified, partially proprietary simulation kernel [5, 6, 22]. Finally, the usefulness of some of the available tools still turns out to be frequently clouded by crashes or unexpected behaviour at the end of the day. As a result, most SystemC users eventually end up in straight code entry using a standard C++ development environment such as Eclipse or KDevelop. Also, since a major advantage of SystemC over other system-level design languages is its open nature, the availability of development tools as open-source or at least their compatibility to the open-source SystemC simulation kernel plays a role in making a decision as well.

In this work, we deal with the question whether SystemC designers can be assisted with transaction-level design and communication analysis by a lightweight and easy-to-use TLM analysis framework which works off-the-shelf with virtually any SystemC simulation kernel and requires only minimal effort to be set up. The main contribution of our work is a set of SystemC classes which provide a very simple and extensible interface to perform the following analysis tasks on any SystemC TLM model without the need for a parser or a modified simulation kernel:

- Reflection of design structure and hierarchy, i.e. the modules, processes, ports, interfaces, and channels in a TLM model,
- SystemC-aware and user-extendable semantic interpretation of the design structure elements in order to hide unnecessary information and help creating intuitive visualizations,
- Recording of transactions in order to watch and analyse data transfers between communicating processes via channels,
- Runtime introspection into transaction data and other objects of interest,
- Streaming of the gathered analysis output to either a file, a database, or an analysis tool.

While several approaches for each of these tasks already exist, they all have in common that either extensive manual source code modifications are necessary [19] or the usage of a parser in combination with a modified simulation kernel (see section 2).

In this paper, a reflection and introspection framework is presented which extracts all relevant information under any LRM-2.1 [18] compliant SystemC kernel, such as the unmodified OSCI open-source kernel. Moreover, our framework requires just minimal source code modifications in the top-level module to be set up, it is able to record any transaction which is performed over an sc_port, and provides an extensible data interface based on XML streaming for sending the collected log data to a visualization client, a file, or a database.

2 Related work

In order to extract interesting data from a SystemC model, two basic approaches can be distinguished: simulation-based and parser-based. While parser-based approaches investigate the source code of a design, the simulation-based methodology performs an interpretation of the code, and most proposals use the open-source OSCI kernel as a base.

Both methodologies have advantages and drawbacks. Parsing SystemC code is a tough job, since SystemC is based on the expressive general purpose language C++ which allows the SystemC designer to apply any artful C++ technique he is aware of. Hence, parser based approaches are the primary solution when it comes to detailed static analysis of a C++ design.

Extraction of the module hierarchy and their interconnect, however, is a very different beast which needs a broader understanding of the model. To this end, other techniques than parsing are required. Thus, all SystemC parsers known to the authors have in common that they either use a combination of source code parser and model simulation/elaboration or, if they leave out the latter, have conceptual limitations. Also, the combined approaches have some restrictions. For example, the parser used in the PINAPA framework [15] is based on gcc and thus detects all SystemC objects as simple classes. Though PINAPA calculates a mapping of the parser's results to the elaborated SystemC design, it is not able to relate dynamic information such as references or pointers to sc_objects, so that the user is in charge of avoiding such constructs.

While PINAPA and also KaSCPar [10] and ViSyC [11] use a combined parser-/simulation-based approach, ParSyc [9], SystemPerl [20] and SystemCXML [3] are pure SystemC parsers. However, KaSCPar in its current state has some limitations according vital C++ constructs, ParSyc relies on an RTL synthesis subset of SystemC, SystemPerl requires user defined hints in the source code, and SystemCXML is limited by the parsing capabilities of Doxygen [7]. ViSyC seems to be a promising approach, since the authors do not disclose any severe restrictions.

When particularly focussing on the extraction of module hierarchy and interconnect information relevant for TLM-based communication analysis (as in our approach), solely simulation-based methodologies are an alternative solution. While any LRM-2.1 compliant simulation kernel provides a lot of design hierarchy data itself (using the sc_get_top_level_objects function [18]), useful transaction recording and data introspection can only be achieved by utilising a modified simulation kernel such as provided with Vista [22] or by annotating the whole design with SystemC Verification Framework [19] extensions. However, in our opinion a solution is required that neither relies on a proprietary kernel nor needs massive code puffing. In the following, we present a simulation-based open-source approach which fulfils these requirements.

3 Definitions

Our approach with the objective of reflecting the design structure of SystemC models along with transaction recording could only be achieved by a meaningful classification of the data of interest. Below, we give such a categorization.

3.1 Design structure

The design structure, as we understand it, mainly consists of the *module hierarchy* as defined in the SystemC 2.1 Language Reference Manual [18] and interfaces (class sc_interface).

Doucet et al. [8] distinguish between three major categories of reflection data. These are

1. Design information (structural and behavioral),
2. Run-time infrastructure information, and
3. Modeling dimension information.

These categories are divided in further subcategories. According to this categorization, the design structure belongs to the subcategories *(i) structural design information* and *(ii) static simulation information*. Transaction data can be recorded by means of *(iii) simulation callbacks*.

Doucet et al. state different ways of collecting reflection information:

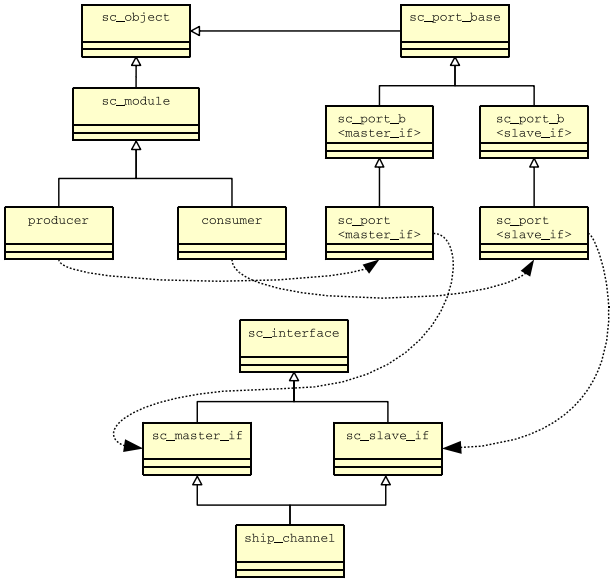
1. The observer pattern in design components,
2. Sub-typing of modeling constructs,
3. Composition replication for introspection,
4. Using a declarative meta-language.

The design structure information consists virtually only of *(ii) static simulation information* with the exception of interfaces which belong to *(i) structural design information*. Additionally, user-defined changes on the SystemC modeling constructs such as sc_module, sc_port, etc. are not of interest in many cases; this means, that e.g. a user-enhanced sc_module implementation based on inheritence in principle should be treated like the standard sc_module.

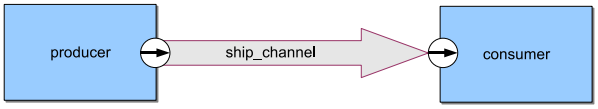
However, a main goal of this work is to not only extract syntactic information from a SystemC model, but also to *interpret* the gained information in a way such that the inherent semantic information is retrieved. Thus, a hierarchical channel, for example, which actually is assembled from several hierarchical modules, should be treated as a single connection component by our framework.

Figure 1 illustrates this approach with an example. While in 1(a), an ordinary UML class view of a SystemC model is given, 1(b) shows a simplified but more informative model derived from adding model-specific transaction-level information to the structure data.

Taking the minimal intrusion requirement into account, we can avoid source code parsing and generation of meta-level information from modeling constructs if we were able to extract all design-relevant



(a) SystemC model class diagram



(b) SystemC model abstract representation showing only design-relevant information

Figure 1: Different SystemC model representations

information belonging to both *(i)* and *(ii)* without (or with only minimal) simulation kernel or user code modifications.

3.2 Transactions

The second important aim of DUST is to enable automatic transaction recording for SystemC models. Transaction recording is a well-known verification technique which is primarily suited for high-level model checking with TLM. The orthogonalized structure inherent in this design paradigm can be easily utilized for transaction recording.

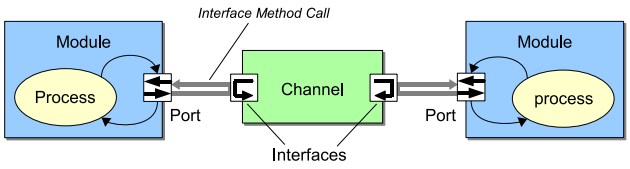


Figure 2: Interface method calls (IMCs) via ports

In SystemC, TLM is based on interface method calls (IMCs), so that all communication between modules is performed by calling interface methods via sc_ports. The interface methods are implemented by channels, which build an interconnect architecture between two or more ports bound to the channel. The data to be transferred through the channel is passed as a parameter when invoking the interface method. Figure 2 illustrates this behavior.

In the following, we consider a transaction to be a sequence of IMCs, delimited by a *start* IMC and an *end* IMC. Depending on the abstraction level of the model, time may pass during a transaction but does not have to. Independend from that, the order of the IMCs inside a transaction always reflects the IMC call sequence of the initiator and hence must not change during transmission. The resulting view of a transaction is illustrated in figure 3.

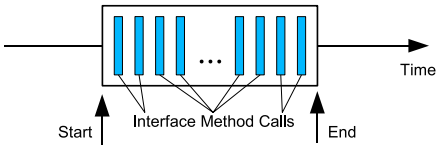


Figure 3: A transaction

To enable adequate transaction analysis and introspection, our framework must provide a way to record all IMCs of a transaction as well as all data objects transported from the transaction initiator to the target as a parameter of these IMCs.

4 DUST framework

In this section we introduce the DUST[1] framework for design structure reflection and transaction recording with SystemC. Besides the minimal-intrusive acquisition of information it aims at data representation and visualization.

Figure 4 shows an outline of our approach. It splits up into the following three main parts:

1. Data sampling and processing,
2. Data representation,
3. Visualization.

The first part is responsible for design structure reflection, transaction recording and XML processing. In the second part, we have defined an XML data format to represent, store and exchange the structural and behavioral information. Third, we have implemented viewers visualizing transaction data in combination with design structure.

The first part has two major tasks. The first one is the reflection of design structure (module hierarchy and interfaces). The second task is performing transaction recording during the simulation phase. We describe these components in the following sections.

4.1 Structure reflection

Most of the design structure information is inherent in the simulation context which is built and managed

[1]Dynamic and Universal SystemC Transaction framework

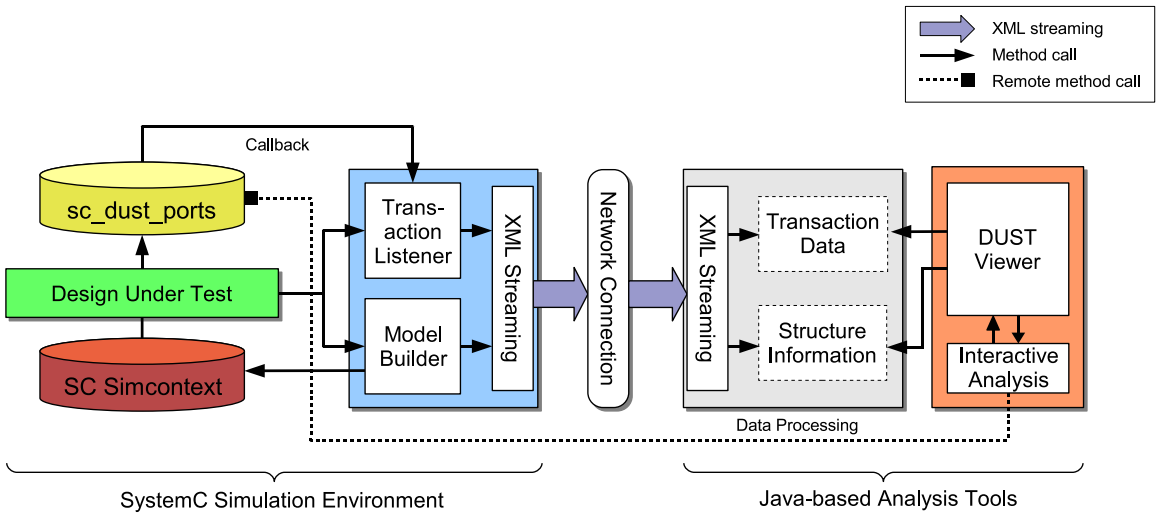


Figure 4: DUST framework overview

by the (OSCI) SystemC kernel. The simulation context serves as a design composition database for the module hierarchy.

The design structure reflection is done at the end of the elaboration phase which is part of the execution of a SystemC application. To this end, we make use of the end_of_elaboration function which is contained in the standard implementation of the classes sc_module, sc_port, sc_export and sc_prim_channel. By default, these functions do nothing. They must be overridden in a user defined subclass of these classes in order to perform some action.

```
void end_of_elaboration() {
    // generate design structure
    model_builder xml(true);
}
```

Listing 1: Building design structure at the end of the simulation kernel's elaboration phase

Listing 1 shows a code snippet of the overridden function which triggers the design structure reflection with DUST. The design structure is extracted from the simulation context. To this end, we call the global simulation kernel function sc_get_top_level_objects. Subsequently, the module hierarchy can be extracted from the simulation context. In order to build the design tree, the basic design elements must be distinguished.

The dynamic_cast operator is used to discover the type of the returned sc_objects. For this purpose, a cast to the bottommost base class of the wanted class which is no template class is performed. Otherwise, the dynamic_cast would always fail since it considers two template classes instantiated with different template parameters as two different classes.

Using a depth-first search, the complete module hierarchy can be extracted and transformed to a suitable storage format subsequently. Hence, we have defined a special XML-format for storing the design structure of SystemC models to be discussed later in the paper.

Interface information

In contrast to the extraction of the module hierarchy, interfaces cannot be distinguished within the simulation context, since they are not derived from the base class sc_object and therefore do neither belong to the module hierarchy nor to the object hierarchy.

To become aware of interfaces, we use the typeid of ports. This class identifier belongs to the runtime type information (RTTI) provided by C++ and contains a type name, which is an implementation dependent string. Template parameters of a template class are part of the type name for this class.

Since ports use template specialization on specific interface classes, type names which are specialized on different interfaces differ. Also, it is known to which channel a port is bound. Thus, by comparing the typeid of the ports bound to a channel, one can obtain the interfaces which are implemented at least by this channel. Using this method, any channel interface to which at least one sc_port is bound can be detected. This approach is outlined in figure 5.

4.2 Transaction recording

To enable automatic transaction recording with DUST, we have to find a way to get aware of every IMC in the model taking place during simulation. Since IMCs in SystemC always result in a call to the sc_port operator −> for accessing the channel which implements the interface and is bound to this port, we propose to infiltrate DUST's transaction recording features into the sc_port class. Using this approach, we are able to meet our design requirement to be *minimal-intrusive* while at the same time also user-defined port implementations which inherit from sc_port can be automatically included in transaction recording. Also, this enables easy integration of DUST into existing SystemC modelling frameworks, such as SysteMoC [21] and GreenBus [14].

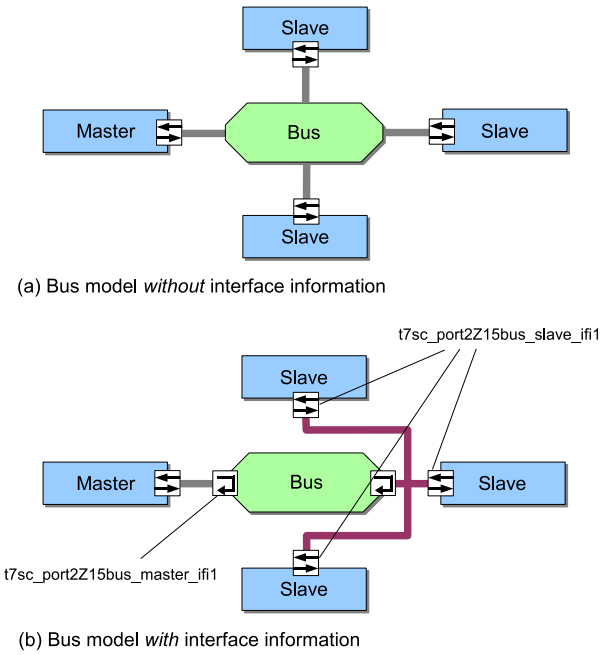


Figure 5: Bus model structure analysis (a) without, and (b) with interface information

In our current implementation, the class sc_port has been enhanced to sc_dust_port in such a way that any calls of an interface function are recorded. Within this class, the operators which are required to perform the IMCs are overridden with new versions which contain transaction recording code. By implementing additional control functions in the sc_dust_port, it is furthermore possible to modify the transaction recording behaviour of the port. To give an example, we have implemented transaction filter functions which can be configured during a running simulation.

The transaction recording itself is done by leveraging the SystemC Verification Library (SCV) [19]. This library allows for transaction based verification and data introspection. However, when applied by the user, SCV requires a considerable amount of additional source code to be added at each position in the model where transaction recording is desired. By using an SCV-enabled sc_dust_port instead, this task can be automated, and redundancy is avoided.

```
// operator function analog to base class
inline IF* operator -> () {
  IF *ret = sc_port<IF, N>::operator -> ();
    ...
  scv_tr_handle h =
    call_gen.begin_transaction(begin_data);
    ...
  call_gen.end_transaction(h, end_data);
  return ret;
}
```

Listing 2: Transaction recording within sc_dust_port

In order to record a transaction, we instantiate an SCV recording database. Additionally, a transaction stream and a generator are created for every sc_dust_port. Recording itself then is done by calling the appropriate recording methods of the generator instance inside the overridden operator functions.

Listing 2 shows an example of an overridden operator function which is responsible for transaction recording. From a user's point of view, the only thing to do is replacing the class sc_port with the advanced version sc_dust_port for any port of interest.

However, this approach has one drawback: Given only the basic functionality to detect arbitrary IMCs on an sc_dust_port, we are not able to detect *start* and *end* timepoints of a transaction (figure 3). To overcome this, the user can indicate the *start* and *end* interface methods of his channel implementation by using the macros DUST_START_IMC and DUST_END_IMC. Otherwise, each IMC will be considered a single transaction.

4.3 XML format

Besides design structure reflection and transaction recording we have specified an XML format for storing this information. It allows for standardized representation, saving and exchange of structural and behavioral information.

```
<xsd:group name="module.component.mix">
 <xsd:choice>
  <xsd:element ref="module"/>
  <xsd:element ref="prim_channel"/>
  <xsd:element ref="port"/>
  <xsd:element ref="process"/>
 </xsd:choice>
</xsd:group>

<xsd:group name="static_model.component.mix"
 <xsd:choice>
  <xsd:element ref="module"/>
  <xsd:element ref="prim_channel"/>
 </xsd:choice>
</xsd:group>

<xsd:element name="static_model"
  type="SystemCModelType"/>
<xsd:complexType name="SystemCModelType">
 <xsd:sequence>
  <xsd:group maxOccurs="unbounded"
      ref="static_model.component.mix"/>
 </xsd:sequence>
</xsd:complexType>

<xsd:element name="module">
 <xsd:complexType>
  <xsd:sequence>
  <xsd:element name="kind" type="xsd:string"/>
  <xsd:element name="name" type="xsd:string"/>
  <xsd:group maxOccurs="unbounded"
   minOccurs="0" ref="module.component.mix"/>
  <xsd:element maxOccurs="unbounded"
   minOccurs="0" ref="interface"/>
  </xsd:sequence>
  <xsd:attribute name="id" type="xsd:ID"
   use="required"/>
 </xsd:complexType>
</xsd:element>
```

Listing 3: Basic topology of XML model for design structure

Our definition is divided into two parts. On the one hand we have a design structure model, on the other hand there is a transaction model containing references to design elements in the structure model.

Since the design structure often contains very structured and hierarchical data, XML is especially well suited for storing this information. We developed a formal definition of the XML format in XML Schema [24]. It consists of two schema documents which contain the design structure and the transaction data. The structure of the former is shown in listing 3.

Besides using a self-defined data format, one option was using the open standard format XML Metadata Interchange (XMI) [17]. XMI is very well-suited for the representation of fully detailed UML models. However, with XMI it is difficult to become aware of the special role certain classes play within a SystemC model. In our data format, we mainly focused on the relevant SystemC specific information inherent in the C++ user model, which has a special semantic inside the model (e.g. port-to-interface binding).

However, if Dust output is to be imported into a standard UML tool, a transformation between our XML format and XMI (and therefore to UML) could be done by adapting the design principles of the standard to our format, which would basically result in changing data types as well as linking. For this purpose, XSLT [25] is a proper solution.

4.4 XML streaming

The output format of Dust is XML. We have implemented an XML streaming interface which enables forwarding the extracted analysis data during a running SystemC simulation to various output filters. Several filters can be used in parallel. For example, the analysis data may be saved to an XML file and can be streamed via a network connection to a remote analysis tool simultaneously.

Thus, both interactive runtime analysis as well as later offline analysis of the recorded simulation data is supported. Finally, for faster data access, Dust data can also be stored in a database, which especially speeds up resolving links between transaction records and structure items.

4.5 Visualization

To enable visual analysis of the design structure and transaction recording data, we developed a Java-based visualization library, which is based on Apache XMLBeans [2] and the Java Foundation Classes library [23].

The visualization library splits up into three parts:

- XML input filters,
- XML data processing and access functions,
- Visualization API.

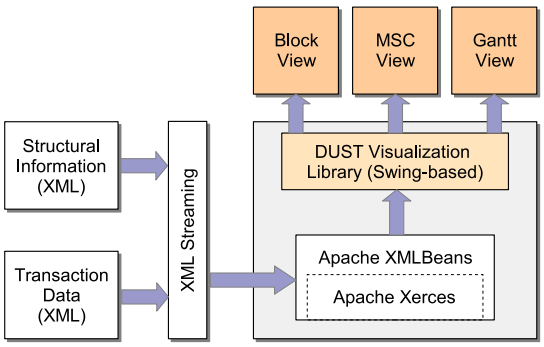


Figure 6: Dust visualization software architecture

The software architecture of this library is outlined in figure 6. In accordance with the XML output filters of the SystemC part of Dust, corresponding input filters have been implemented for establishing a direct network link to the SystemC simulation as well as reading XML data from a file.

To gain access to the individual XML elements, we read the recorded structure and transaction data with the aid of the Apache XMLBeans [2] package which relieves the developer of defining various data types as well as get- and set-methods for representation and accessing XML elements and attributes. XMLBeans generates class definitions from XML Schema definitions and then provides JavaBeans-style accessors.

Based on these accessors, a visualization API has been developed which allows for easy access to design structure information and transaction data. Additionally, basic analysis functions are provided for applying filter functions to transaction data, mapping transaction data to design structure elements, and averaging channel workload over a period of time.

5 Case examples

In this section we illustrate the capabilities of Dust by two example designs. The SystemC designs we have chosen are a motion detection processor and a JPEG encoder.

Motion detection

The motion detection processor is a System-on-Chip that processes video frames in real-time and calculates regions where motions take place.

A high-level schematic of the motion detection processor is given in figure 7. The video processing starts in the module CAM_TO_YUV which receives video data from a camera device. The MORPH modules and DETECT_REGION handle the motion detection. The module DISPLAY forms the endpoint of the processing and displays the video frames overlayed by detected regions of interest. SHIP channels [13] serve as transaction-level communication connection.

The system consists of a flat system model. This means that the modules contained do not hold any submodules or subchannels.

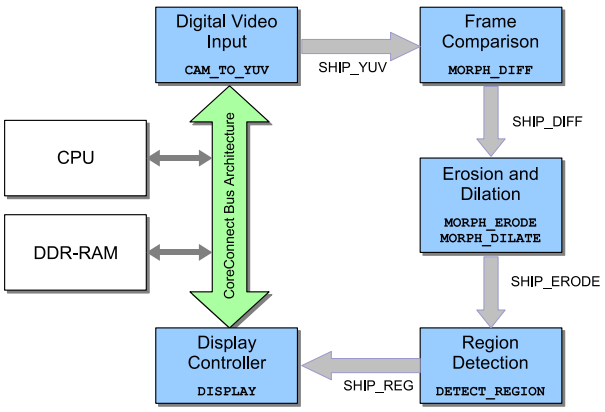


Figure 7: Motion detection block diagram

Naturally, the motion detection generates a huge amount of data during a short period of time. The simulation of 100 video frames with the original SystemC model takes 5.6 seconds on our test system (Athlon Opteron 800 with 2.2GHz, Linux 2.6). With DUST transaction recording *and* transaction introspection applied, 9.3MB of transaction data per processed video frame are generated and the simulation slows down to 234 seconds. However, if introspection is not necessary, only 173KB of transaction XML data per processed video frame are generated and the DUST-enabled simulation performs at almost the same speed (5.7 seconds) as the original simulation without DUST.

Figure 8 shows examples of DUST viewers based on our visualization library. Figure 8(a) displays a design structure of the motion detection processor. Figure 8(b) shows a sequence of transactions which were recorded during the simulation of this model.

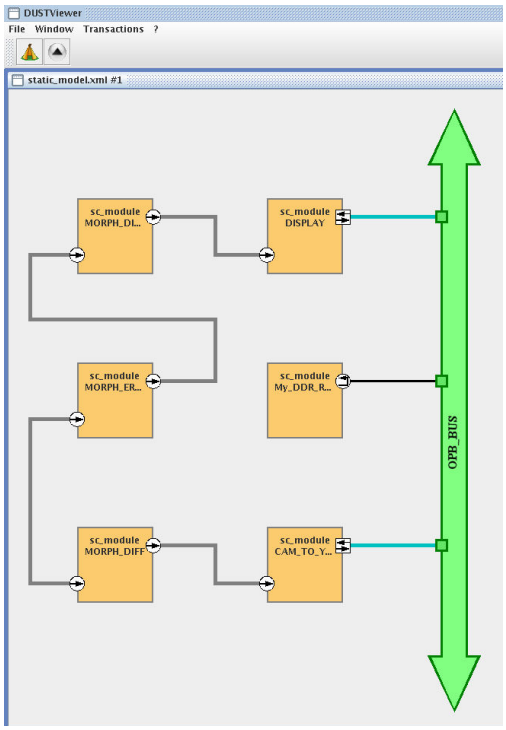
JPEG encoder

SystemC supports the defining of strongly hierarchical models. Our second example, the JPEG encoder, is a typical example for this class of models and demonstrates the capability of our system to reflect hierarchical models. One path through the module hierarchy of the JPEG encoder is presented in figure 8(c). Section I shows the model's top level module, II to VI show lower layers of the module hierarchy, which are contained in the top level module.

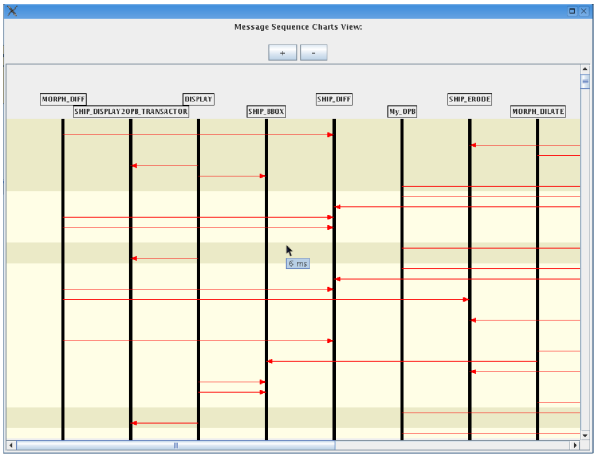
6 Discussion of our approach

Traditional verification methods such as signal tracing and conventional debugging turn out to be all but easy to use for todays large designs containing many parallel communicating processes.

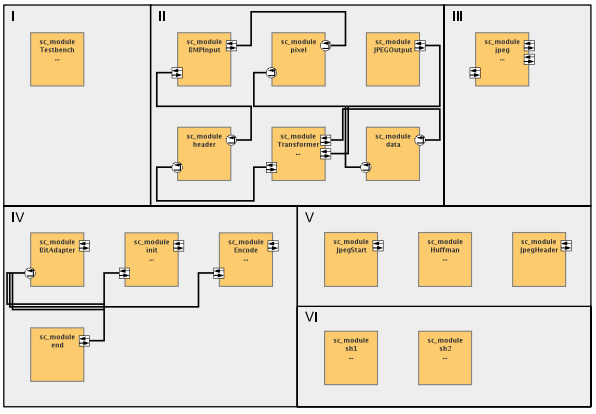
We have shown that it is not always required to parse the model's source code or to alter the SystemC simulation kernel in order to obtain design structure and transaction recording information. In particular, we demonstrated that it is possible to gain the full module hierarchy along with information about interfaces from the simulation context. Moreover, our



(a) Structure view of the motion detection processor



(b) Message sequence chart of the motion detection processor



(c) JPEG encoder's module hierarchy

Figure 8: DUST viewer examples

approach is capable of breaking down the extracted structure and simulation data to an optimized representation which reflects only the information really relevant to the designer. However, there could remain some unbound interfaces which cannot be detected with our method.

As experience with several student projects shows, our approach is very easy to use for automated transaction recording without the need for learning and applying the SystemC Verification Library to one's source code. In combination with a bus simulation framework such as GreenBus [14], the simulation results can quickly be analysed in-depth, and better system configurations can be achieved faster.

Nevertheless, it should be pointed out that we do not propose our framework as a replacement for fine-grained verification frameworks such as PINAPA [15] or ViSyC [11]. As an advantage compared to DUST, these parser-based approaches also enable cross probing (i.e., switching back and forth between source code and visualization), which is essential for debugging a faulty design. DUST's key field of application explicitly is communication analysis of transaction-level models.

7 Conclusion

In this paper, we presented a structure reflection and transaction recording framework including an XML data format definition and a visualization library which gives SystemC designers the opportunity to obtain valuable information about communication inside TLM models.

Our approach requires minimal changes to the project source code in order to enable both design structure extraction and transaction recording. The proposed framework is fully based on open-source solutions and works with any LRM-2.1 compliant SystemC kernel. It will be available for download on the GreenSocs open source platform www.greensocs.com soon.

In our ongoing work, we are experimenting with a runtime-configurable sc_dust_port which allows for simulation speed control and enables interactive fault injection into a running simulation.

References

[1] Actis Design. AccurateC: Static C++ Code Analysis for SystemC. *Actis Design, Portland*, 2005.

[2] Apache Software Foundation. XMLBeans Homepage. *http://xmlbeans.apache.org*.

[3] D. Berner, H. Patel, D. Matthaikutty, J.-P. Talpin, and S. Shukla. SystemCXML: An Extensible SystemC Front End Using XML. *Proc. Forum on Design Languages (FDL)*, Lausanne, September 2005.

[4] L. Cai and D. Gajski. Transaction Level Modeling: An Overview. *Proc. CODES+ISSS*, 2003.

[5] Celoxica. Agility Manual for Agility 1.1. *Available at http://www.celoxica.com*.

[6] CoWare Inc. ConvergenSC Homepage. *http://www.coware.com*.

[7] D. van Heesch. Doxygen Homepage. *http://www.doxygen.org*.

[8] F. Doucet, S. Shukla, and R. Gupta. Introspection in System-Level Language Frameworks: Meta-level vs. Integrated. *Proc. Design, Automation and Test in Europe Conference (DATE)*, München, March 2003.

[9] G. Fey, D. Große, T. Cassens, C. Genz, T. Warode, and R. Drechsler. ParSyC: An Efficient SystemC Parser. *Proc. 12th Workshop on Synthesis and System Integration of Mixed Information Technologies (SASIMI)*, Kanazawa, October 2004.

[10] FZI. KaSCPar - Karlsruhe SystemC Parser Documentation. *Available at http://www.fzi.de*.

[11] C. Genz and R. Drechsler. System Exploration of SystemC Designs. *IEEE Computer Society Annual Symposium on VLSI*, Karlsuhe, March 2006.

[12] F. Ghenassia. *Transaction-Level Modeling with SystemC*. Kluwer Academic Publishers, 2006.

[13] W. Klingauf. Systematic Transaction Level Modeling of Embedded Systems with SystemC. *Proc. Design, Automation and Test in Europe Conference (DATE)*, München, March 2005.

[14] W. Klingauf, R. Günzel, O. Bringmann, P. Parfuntseu, and M. Burton. GreenBus - A Generic Interconnect Fabric for Transaction Level Modelling. *Proc. Design Automation Conference (DAC)*, San Francisco, CA, July 2006.

[15] M. Moy, F. Maraninchi, and L. Maillet-Contoz. PINAPA: An Extraction Tool for SystemC descriptions of System-on-a-Chip. *Proc. ACM Int. Conf. on Embedded Software (EMSOFT)*, Jersey City, NJ, September 2005.

[16] W. Müller, W. Rosenstiel, and J. Ruf. *SystemC: Methodologies and Applications*. Kluwer Academic Publishers, 2003.

[17] Object Management Group. MOF 2.0 / XMI Mapping Specification v2.1, September 2005. *Available at http://www.omg.org*.

[18] Open SystemC Initiative. SystemC 2.1 Language Reference Manual. OSCI, April 2005. *Available at http://www.systemc.org*.

[19] OSCI SystemC Verification Working Group. SystemC Verification Standard Specification. OSCI, May 2004. *Available at http://www.systemc.org*.

[20] W. Snyder. SystemPerl Homepage. *http://www.veripool.com/systemperl.html*.

[21] M. Streubühr, J. Falk, C. Haubelt, J. Teich, R. Dorsch, and T. Schlipf. Task-Accurate Performance Modeling in SystemC for Real-Time Multi-Processor Architectures. *Proc. Design, Automation and Test in Europe Conference (DATE)*, München, March 2006.

[22] Summit Design, Inc. Vista System Design and Verification Platform Online Documentation. *Summit Design*, 2006.

[23] Sun Microsystems. Java 2 Platform Standard Edition 5.0 API Specification, 2006. *Available at http://java.sun.com/j2se/1.5.0/docs/api*.

[24] W3C. XML Schema Homepage. *http://www.w3.org/XML/Schema*.

[25] W3C. XSLT Homepage. *http://www.w3.org/TR/xslt*.