

CARH*: A Service Oriented Architecture for Validating System Level Designs

Hiren D. Patel, Deepak A. Matthaikutty, David Berner, Sandeep K. Shukla,

Abstract—Existing system level design languages and frameworks mainly provide a modeling and a simulation framework. However, there is an increasing demand for supporting tools to aid designers in quick and faster design space and architectural exploration. As a result, numerous tools such as integrated development environments and others that help in debugging, visualization, validation and verification are commonly employed by designers. As with most tools, they are targeted for a specific purpose, making it difficult for designers to possess all desired features from one particular tool. Only public-domain tools can be easily extended or interfaced with other existing tools, which a lot of the existing commercial tools do not promote. Having an extendable framework allows designers to implement their own desirable features and incorporate them into their framework. However, for technology reuse and transfer, it is important to have a tidy infrastructure for interfacing the extension with the framework, such that the added solution is not highly coupled with the environment making distribution and deployment to other frameworks difficult if not impossible. This requires a plug-and-play framework where features can be easily integrated. In this paper, we tackle these issues of extendibility, deployment, the inadequacies in system level design languages and frameworks by presenting a service oriented architecture for validating system level designs for SystemC called CARH, that uses a variety of open-source technologies such as Doxygen, Apache’s Xerces XML parsers, SystemC, TAO and ACE.

Index Terms—SystemC, CORBA, Middleware, Reflection, Inspection, Verification and Validation, Embedded System Design

I. INTRODUCTION

The rising complexity of embedded system design and the increasing engineering efforts for their realization has resulted in a widening of the productivity gap. Efforts towards mitigating the productivity gap have raised the importance of System Level Design (SLD)s languages and frameworks. In recent years, we have seen SLD languages such as SystemC, SpecC, SystemVerilog [1], [2], [3] in efforts to raise the level of abstraction in hardware design. These SLDs assist designers in modeling, simulation and validation of complex designs. However, the overbearing complexity and heterogeneity of designs make it difficult for embedded system designers to meet the time-to-market with just the provided SLDs. Hence the proliferation of numerous commercial tools supporting SLD languages with features for improved model building experience. This shows that designers need improved techniques, methodologies and tools for better debugging, visualization, validation and verification in order to improve productivity.

Most of the present SLD languages and frameworks (SLDL) are primarily equipped with a modeling and simulation environment and lack facilities to ease model building, visualization, execution analysis, automated test generation, improved debugging, etc., which are becoming increasingly important to enhance the overall design process. As evidenced by the release of commercial tools such as Debussy [4], ConvergenSC [5], VCS [6], Incisive [7] to mention a few, SLDL themselves require additional supporting tools. However, with the multitude of commercial tools, a designer must select the set of tools that best fits his/her needs and many times the existing tools do not suffice. This raises an issue of extendibility, where designers have no way to extend the commercial tools or solutions to suit their own specific needs. Even if the SLDL can be subject to alterations, the abundance of open-source tools are subdued by the issue of efficient deployment mechanisms and reusability. In addition to the ability of designers to extend a framework, they should also be able to cleanly interface and distribute their solution for integration with other SLDLs.

Some industrial outcomes to addressing these inadequacies in SLDLs are simulation-based dynamic validation frameworks such as ConvergenSC [5], VCS [6] and Cadence Incisive [7]. However each of these tackle various aspects of the design process. There are some overlapping features and some distinct desirable features, and at this moment it is not possible to unify the capabilities into one framework. Furthermore, the inherent problem of extendibility, disallows users to consider altering or integrating these different frameworks.

In efforts to address these issues with industrial simulation-based dynamic validation frameworks and the lack of features for SLDLs, we propose a simulation based dynamic validation framework called CARH as a solution for SystemC. CARH is a service oriented verification and validation framework using middleware technology. In particular we employ TAO [8] that is real-time (R/T) CORBA Object Request Broker (ORB) [9] and ACE [10], to allow for a language independent pluggable interface with the framework. We show the use of public domain tools, Doxygen, XML and Xerces-C++ for structural reflection of SystemC models, thus avoiding using front-end parsing tools such as EDG [11]. We exploit the open-source nature of SystemC to perform runtime reflection. We also introduce services for improved debugging, automated test generation, coverage monitoring, logging and a reflection service. ACE also provides design pattern implementations for multi-threaded models, thread pooling and synchronization. The inherent R/T facilities from TAO allows for R/T software modeling. This is also ideal for cosimulation, because it fa-

* We code name our software systems after famous computer scientists. CARH (*kär*) stands for C. A. R. Hoare.

enables independent languages that support CORBA to easily interface with CARH. In CARH we have built a R/T middleware based infrastructure using Event service and Naming service among CORBA services, and by building a number of domain specific facilities such as reflection, automated test generation and dynamic-value change dump (d-VCD).

A. Main Contributions

We enlist the fundamental contributions of our work:

- Service-orientation a necessary approach for allowing a multitude of features that enable multi-platform debugging, visualization, performance and coverage monitoring for system level designs.
- The use of a standardized interface-based mechanism for allowing the different services to communicate and exchange information amongst themselves. Using the OMG standardization of CORBA-variant implementations, any new feature that follows this standardization can be easily integrated into CARH.
- Introspective facilities for the models: Facilitating any such capabilities requires that the infrastructure have the ability to introspect the models. Our reflection and introspection mechanism improves debugging capability by providing automated test generation and extraction of runtime information of SystemC models.
- Debugging facilities: Unfortunately, standard debuggers prove less useful when debugging multi-threaded models and we need to facilitate the designer with model execution information in terms of call graphs, dynamic-Value Change Dump (d-VCD) and logging facilities.
- Automated test generation capabilities: Even though SCV has utilities for randomized constraint and unconstrained based test generation, there is no infrastructure that automatically generates testbenches for particular models and stimulates them.

B. Organization

In Section 2, we briefly discuss the inadequacies of SLDLs and current dynamic validation frameworks. The following section outlines our approach to addressing these inadequacies. In Section 4, we discuss related work with Reflection and Introspection (R-I) along with the technologies we employ in endowing SystemC with R-I. Section 5 presents the CARH architecture followed by a detailed description of the services rendered. Section 7, describes the usage model for the CARH framework. In Section 8, we provide simulation result for the FIR and FFT modeled using CARH framework and finally concluding remarks and future work in Section 9.

II. ISSUES AND INADEQUACIES OF CURRENT SLDLs AND DYNAMIC VALIDATION FRAMEWORKS

There are numerous SLDLs employed in the industry such as Verilog, SystemVerilog, SpecC, VHDL and SystemC [3], [2], [12], [1] that primarily comprise of two aspects. The first being a modeling framework and the other being a simulation framework. The modeling framework allows

designers to express designs either in a programmatic or graphical manner and the simulation framework is responsible for correctly simulating the expressed model. Designers can create models using the modeling framework and verify the design via simulation. However, with the rising complexity in current designs, it is not enough to simply provide designers with a modeling and simulation framework. It is becoming evident that designers require supporting tools to increase their productivity and reduce their design time. We enlist some of the important supporting features that are necessary for today's designers. They are: introspection in SLD languages and frameworks, automated testbench generation, coverage monitors, performance analysis and enhanced visualizations.

There are several industrial solutions that are targeting problem areas in SLDLs by providing some of these above mentioned features. However, instead of describing the multitude of commercial solutions for some of the inadequacies with SLDLs, we discuss some of the validation frameworks that are commercially available for improving design experience. Some of them consist of the features that we realize as being important for SLDLs.

Few of the popular validation framework solutions are ConvergenSC [5], VCS [6] and Cadence Incisive [7]. Each one of these tools have good solutions for different aspects in aiding designers for improved validation and model building experience. For example, ConvergenSC presents a SystemC integrated development environment (IDE) using Eclipse. This IDE has project management support, version control and build support. It also has an advanced debugging environment allowing step-wise execution that is specially geared towards QuickThreads used in SystemC. Although ConvergenSC provides a good IDE for improving design productivity and debugging, it does not support any testbench generation features that are crucial for correctness of designs. On the other hand, VCS targets the aspect of RTL verification by incorporating multi-language support for Verilog, VHDL, SystemVerilog and SystemC, coverage techniques for Verilog and mixed-HDL designs, assertion-based design verification, and testbench generation constructs. It can interface with other Synopsys products. Similarly, Incisive from Cadence also supports some of the similar features for integrated transaction environment, unified simulation and debug environment, assertion support and testbench generation. By simply looking at some of these commercial solutions, we notice that neither one of these tools fully suffice the needs of a designer.

One apparent and a major drawback in the above mentioned industrial solutions is that of extensibility. Due to the commercial nature of these tools, it is impossible for designers to extend the existing framework themselves. Even though ConvergenSC's Eclipse IDE allows for plugins, their debugging facilities may not be easily extendable as desired by users. Furthermore, none of these solutions are open-source, disallowing users to consider alterations. Hence, designers have to look elsewhere for technology that can complement their existing solution. None of these tools can satisfy every designer's requirements thus necessitating the use of a variety of additional tools. For example, a designer may require ConvergenSC as an IDE, but also has to use VCS for RTL

verification along with System Studio for system level design support. Even then, the use of multiple commercial tools may still not satisfy a specific purpose in the mind of the designer. This difficulty can be overcome by providing a framework that is a part of the public-domain, and that is easily extendable.

Another important concern is the deployment and reuse of technology. Often times, there are designers who create specific tools to perform certain tasks that are difficult to distribute because the tools are highly coupled with their environment. For example, suppose some designer implements a hot-spot analyzer for ConvergenSC as a plugin. This plugin would probably be highly coupled with the Eclipse and ConvergenSC environment making it difficult to adapt to other IDEs or environments. Therefore, another designer using a different environment may have difficulty in interfacing with this plugin without using that particular set of tools. Hence, it is important that extensibility is followed by a clean deployment mechanism so that plugins interact with their environment through an interface such that an easy adaptation to a different third party environment is possible. A well constructed solution for deployment also promotes unification of relevant tools with a clean interfacing mechanism that facilitates seamless interoperability between multiple tools.

III. OUR GENERIC APPROACH TO ADDRESSING THESE INADEQUACIES

We propose a generic approach that addresses the primary inadequacies of extensibility, deployment and reuse in existing validation frameworks and features for supporting tools that we perceive as being important for SLD languages and frameworks.

A. Service Orientation

Our approach promotes the idea of service orientation in SLDs, where features are implemented as services that interact with each other through IDL interfaces that are language neutral interfaces defined by OMG [13]. Thus, the entire architecture comprises of multiple floating services that can be queried by clients through a common interface. Furthermore, this solution must be part of the public-domain such that designers may add services at their will.

Extendability comes naturally with such an approach because a particular feature can simply be integrated into the architecture as a service that can also employ other existing services. In addition, deployment is relatively easy because the newly added service follows a strict interface convention that it must adhere. Lastly, the implementation of the feature can be independent of the interface as a service, allowing easy distribution and reuse of the feature.

Furthermore, features from different language paradigms may be integrated with the architecture as long as the language can interact with the interface mechanism. This means multi-language services can be integrated into the architecture through this service/interface mechanism. For example, a visual interface displaying performance graphs and tables may use Java as the core language. However, the actual simulation is performed in a C++ based environment and the interface

mechanism allows for the two to communicate and exchange messages.

The advantage of following the OMG standardization, which TAO or any other CORBA-variant adheres to, is that all implementations of OMG standards can be seamlessly used with CARH. A commercial tool that can conform to such a standard allows for easy integration and adoption to CARH.

In addition, these features can be distributed over multiple computers or networks promoting a distributed simulation and validation infrastructure. An example of using a distributed framework is described in [14], that allows compilation and execution of testbenches on a distributed network.

B. Introspection Architecture

The use of introspection is commonly seen in programming languages such as Java and languages that use the .NET framework. However, infiltrating introspective capabilities in SLDs is still a sought after feature. Most attempts at introspection in SLDs are facilitated via structural reflection, but runtime reflection also offers even more possibilities. If an SLD inherits introspective capabilities, then projects such as IDEs, value-change dump (VCD) viewers, improved debugging, and call graphs could take advantage of the reflected information. Unfortunately, there are very few nonintrusive methods for structural reflection and hardly any for runtime reflection [15].

C. Test Generation and Coverage Monitor

Most designers are also responsible for constructing their testbenches that perform sanity and functional tests on their designs. Manual testbench generation is tedious and many times randomized testbenches, weighted testbenches, range-based testbenches are sufficient in validating functional correctness. Furthermore, a testbench generator can take advantage of an introspective architecture to query information regarding ports, signals, types, bitwidths, etc. and then automatically with some user-hints generate testbenches. However, most SLDs do not come with a facility to automatically generate testbenches. Thus, making it another important supporting feature for SLDs.

Coverage monitors provide a measure of the completeness of a set of tests. Computing coverage during validation requires runtime information. This is where the runtime reflection capability can be exploited. In this paper we do not discuss about specific algorithms for test generation or coverage driven test generation because this paper is about the service oriented validation framework and these are but a few examples of services we need to integrate in such an environment. So the interfaces of these services are relevant to this paper and not the algorithms themselves.

D. Performance Analysis

As with most large designs, simulation efficiency is usually a major concern for designers for timely validation purposes. For this, performance analysis features are essential to SLDs. There are designers who require hot-spot analysis to identify the bottlenecks of the design such that they can focus their

optimization techniques towards that section. There are numerous metrics for measuring the time consuming blocks. For hardware design languages using a discrete-event simulation, we envision a performance analysis service that provides the designer with the following capabilities:

- The amount of time every hardware block or module consumes.
- The time spent in computation per module versus inter-module communication time.
- The number of times a particular block is scheduled to execute.
- The frequency of delta events and timed events generated by modules.
- Designer specified timers for identifying time taken in particular sections of the implementation.
- Version comparisons such that altered versions of the design can be aligned with the previous versions and performance metrics can be easily compared through graphs and tables.

These are some of the many capabilities that we see important for performance comparisons. However, performance analysis features are crucial in SLDLs for improving simulation efficiency of the designs, thus an important required feature for SLDLs.

E. Visualization

When working with large designs, a visual representation can be helpful in many ways. Visualization is an important tool for the designer to keep an overview and better manage the design. Using a graphical representation than requiring designers to traverse through many lines of code can immensely benefit design experience. Especially during architectural exploration and refinement, visualizations can help to take important decisions or to reveal problematic areas of the design. Visualizations for communication hot-spots, code usage, or online control flow analysis can give fast intuitive information about important key figures of the design.

In order to obtain meaningful and visually appealing graphs, we need two things: (i) The required data has to be collected and be made easily available. This can be rather obvious for static information such as the netlist or the module hierarchy, but may require important infrastructure support for dynamic information such as communication load or code coverage. (ii) The data has to be processed into visual graphs. This can be a very complex task depending on the type of graph and the level of desired flexibility and interaction. However there are many existing libraries and toolkits that can be used to render a graph structure. The Debussy nSchema and nWave modules [4] for example, are commercial tools for waveform and structural layout visualization. The GraphViz package [16] is an example for a comprehensive graph visualization package that can render and display different types of graphs such as simple textual description and others. Since SLD languages do not come with tools providing such visualizations, we see an important need to add this infrastructure to SLD toolkits in order to take advantage of advanced SLD features.

Visualization in an SLDL toolkit can use data available from other services such as the coverage monitors, introspection architecture, and performance analysis and aid the designer in better comprehending this data. The more services an SLDL offers the more possibilities are there. Visualization can help to take design decisions based on more information in less time. The authors of [17] report an approach for interfacing visualization GUIs to SystemC. However, with the reflection capabilities in CARH we can provide a similar interface in an easier manner.

Until now we discussed the role of SLDLs in modeling and simulation, the need for supporting tools for SLDLs and a good infrastructure for deployment, extendibility and reuse. Now, we present CARH, a service oriented framework for validation of SystemC models. The SLDLs we choose for our experimentation is SystemC [1] and employ the TAO [8] and ACE [10] libraries for the service orientation and implement additional services to promote the supporting features.

IV. BACKGROUND & RELATED WORK

In this section we define reflection and introspection followed by descriptions of some frameworks and languages that provide R-I along with the open-source tools that we employ in deriving our solution for R-I and CARH.

A. Reflection and Introspection

Introspection is the ability of an executable system to query internal descriptions of itself through some *reflective* mechanism. The reflection mechanism exposes the structural and runtime characteristics of the system and stores it in a data structure. We call data stored in this data structure *meta-data*. This data structure is used to query the requested internal characteristics. The two sub-categories of the reflection meta-data are structural and runtime. Structural reflection refers to descriptions of the structure of a system. For SystemC, structural reflection implies module name, port types and names, signal types and names, bitwidths, netlist and hierarchy information. On the other hand runtime reflection exposes dynamic information such as the number of invocations of a particular process, the number of events generated for a particular module and so on. An infrastructure that provides for R-I (either structural or runtime reflection) is what we term an reflection service.

B. Existing tools for structural reflection

Several tools may be used for implementing structural reflection in SystemC. Some of these are SystemPerl [18], EDG [11], or C++ as in the BALBOA framework [19] and Pinapa [20]. However, each of these approaches have their own drawbacks. For instance, SystemPerl requires the user to add certain hints into the source file and although it yields all SystemC structural information, it does not handle all C++ constructs. EDG is a commercial front-end parser that parses C/C++ into a data structure, which can then be used to interpret SystemC constructs. However, interpretation of SystemC constructs is a complex and time consuming task,

plus EDG is not available in the public domain. BALBOA implements its own reflection mechanism in C++ which again only handles a small subset of the SystemC language. Pinapa is a new front-end for SystemC that offers an intrusive solution for parsing SystemC by altering GCC and SystemC's source code. As for runtime reflection, to our knowledge, there is no framework that exposes runtime characteristics of SystemC models.

C. ESys.NET Framework and Introspection in SystemC

ESys.NET [21] is a system level modeling and simulation environment using the .NET framework and C# language. This allows ESys.NET to leverage the threading model, unified type system and garbage collection along with interoperability with web services and XML or CIL representations. They propose the managed nature of C# as an easier memory management solution with a simpler programming paradigm than languages such as C or C++ and use the inherent introspective capabilities in the .NET framework for quicker debugging. They also employ the common intermediate language (CIL) as a possible standard intermediate format for model representation. One of the major disadvantages of using the .NET framework is that it is platform dependent. The .NET framework is primarily a Microsoft solution and making it difficult for many industries to adopt technology built using the .NET architecture because of well-established Unix/Unix-variant industrial technologies.

There are obvious advantages in making ESys.NET a complete mixed-language modeling framework interoperable with SDLs such as SystemC. However, we see no easy solution for interoperability between managed and unmanaged frameworks partly because integrating the unmanaged project in a managed project reduces the capabilities of the .NET architecture. For example, mixing managed and unmanaged projects does not allow the use of .NET's introspection capabilities for the unmanaged sections of the project. A natural way to interact between different language paradigms and development approaches (managed versus unmanaged) is to interface through a service oriented architecture. Microsoft has their own proprietary solution for this such as COM, DCOM and .NET's framework. Unfortunately, one of the major drawback as mentioned earlier is that C# and .NET framework is proprietary technology of Microsoft. Even though there are open-source attempts at imitating C#, the .NET framework as a whole may be difficult to conceive in the near future [22].

The authors of [21], inspired by the .NET framework's reflection mechanism propose the idea of a composite design pattern for unification of datatypes for SystemC. They enhance SystemC's datatype library by implementing the design pattern with additional C++ classes. This altered datatype library introduces member functions that provide introspection capabilities for the particular datatypes. However, this requires altering the datatype library and altering the original source code to extract structural information. This raises issues with maintainability with version changes, updates and standard changes due to the highly coupled solution for introspection.

A different approach for exposing information about SystemC models to graphical user interfaces (GUI) is described

in [17]. This work describes a methodology for interfacing SystemC with external third party tools where they focus on a GUI value-change dump viewer as the external tool. This requires allowing the VCD viewer to probe into the SystemC model and display the timestamp, the type of the signal and the current value of the signal. The authors document the required changes to the SystemC scheduler `sc_simcontext`, `sc_signal` and `sc_signal_base` classes along with their additional interface classes to expose the type of a SystemC signal and its corresponding value. They implement the observer pattern such that the VCD viewer accepts the messages from the altered SystemC source and correctly displays the output.

D. BALBOA Framework

The BALBOA [19] framework describes a framework for component composition, but in order to accomplish that, they require R-I capability of their components. They also discuss some introspection mechanisms and whether it is better to implement R-I at a meta-layer or within the language itself. We limit our discussion to only the approach used to provide R-I in BALBOA.

BALBOA uses their BIDL (BALBOA interface description language) to describe components, very similar to CORBA IDLs [9]. Originally IDLs provide the system with type information, but BALBOA extends this further by providing structural information about the component such as ports, port sizes, number of processes, etc. This information is stored at a meta-layer (a data structure representing the reflected characteristics). BALBOA forces system designers to enter meta-data through BIDL, which is inconvenient. Our method only needs pre-processing of SystemC models. Another limitation of this framework is that the BIDL had to be implemented. Furthermore, the designer writes the BIDL for specifying the reflected structure information which can be retrieved automatically from SystemC source. BALBOA also does not perform runtime reflection.

E. Java, C# .NET Framework, C++ RTTI

Here, we discuss some existing languages and frameworks that use the R-I capabilities. They are Java, C# and the .NET framework and C++ RTTI. Java's reflection package `java.lang.reflect` and .NET's reflection library `System.Reflection` are excellent examples of existing R-I concept implementations. Both of these supply the programmer with similar features such as the type of an object, member functions and data members of the class. They also follow a similar technique in providing R-I, so we take the C# language with .NET framework as an example and discuss in brief their approach. C#'s compiler stores class characteristics such as attributes during compilation as meta-data. A data structure reads the meta-data information and allows queries through the `System.Reflection` library. In this R-I infrastructure, the compiler performs the reflection and the data structure provides mechanisms for introspection.

C++'s runtime type identification (RTTI) is a mechanism for retrieving object types during execution of the program.

Some of the RTTI facilities could be used to implement R-I, but RTTI in general is limited in that it is difficult to extract all necessary structural SystemC information by simply using RTTI. Furthermore, RTTI requires adding RTTI-specific code within either the model, or the SystemC source and RTTI is known to significantly degrade performance.

F. Doxygen, XML, Apache's Xerces-C++

Two main technologies we employ in our solution for R-I for SystemC are Doxygen and XML. Doxygen [23] is a documentation system primarily for C/C++, but has extensions for other languages. Since SystemC is simply a library of C++ classes, it is ideal to use Doxygen's parsing of C/C++ structures and constructs to generate XML representations of the model. In essence Doxygen does most of the difficult work in tagging constructs and also documenting the source code in a well-formed XML representation. Using XML parsers from Apache's Xerces-C++ we parse the Doxygen XML output files and obtain any information about the original C++/SystemC source.

G. TAO and ACE

TAO [8] is a real-time CORBA implementation using the ACE [10] environment. ACE is a library of C++ classes that implement design patterns with focus on network application programming. TAO and ACE together facilitate the user with CORBA and design pattern capabilities.

H. Service Oriented Software

Many distributed applications use middleware such as CORBA [9] to integrate a system with services and floating objects accessible via ORB. System level design languages can take advantage of middleware for cosimulation purposes as shown in [24]. [25] discusses a cosimulation environment for SystemC and NS-2 [26] which can also be integrated into CARH with relative ease. In addition effective testing and parallel simulation execution is viable as demonstrated by [14]. CARH utilizes the TAO & ACE environments to provide a service oriented architecture extendable for cosimulation, distributed testing and any user-desired services.

V. CARH'S SOFTWARE ARCHITECTURE

The architectural description in Figure 1 serves as an extendable road map for CARH. The services and the SystemC models (SystemC-V) are the two main separations in Figure 1. The distinguishing arrows show interaction between the services and the SystemC model. The primary interaction is using an ORB to access services other than the d-VCD that happens to use TCP/IP protocols for communication. First we briefly describe the services within CARH:

Reflection: This consists of two sub-services, where the first exposes structural information about the model and similarly the second exposes runtime/behavioral information about the model.

Testbench: This service automatically generates SCV-based testbenches for the SystemC model using the introspection

capabilities provided by CARH.

Coverage: This performs coverage analysis on collected run-time information and results from simulation to regenerate better testbenches.

Logger: It allows logging of runtime information based on logging requests entered through the user console.

d-VCD: The d-VCD service is independent of an ORB and communicates through network communication protocols. The test generation, coverage monitor and logger services require the reflection and CORBA services. These services can be configured and invoked by a user console. It provides a dynamic-Value Change Dump on the reflected module along with the processes on the SystemC runlist.

The services below are employed by CARH but implemented in CORBA:

Naming: Allows a name to be associated with an object that can also be queried by other services to resolve that name and return the associated object.

Event: Allows decoupled communication between the requestors/clients and the services via an event based mechanism. A clean method to implement push and pull interactions between clients and services.

The remainder elements and facilities of the architecture shown in Figure 1 are described below:

SystemC-V: We implement SystemC-V that supports the SystemC Verification library (SCV) [1] and it also contains an extended version of SystemC 2.0.1 that communicates on an ORB and presents runtime information for introspective clients/services.

The ORB: The OMG CORBA [9] standard distinguishes between horizontal CORBA services, and vertical CORBA facilities. According to that terminology, only CORBA horizontal services we use from TAO are Event and Naming services. On the other hand the reflection, logger, test generation and coverage monitoring services are implemented by us, and qualify as domain specific vertical services.

Console: CARH is operated via the client. The client is a text-based interface that has commands to perform tasks for the user such as startup services, set specific service related flags, specify models to execute and so on.

One of the main strengths of CARH is the possibility of extensions. Developers can write TAO-based services and easily interact with the existing architecture via the interfaces defined. This opens up the possibility of an architecture that can be extended to support what is described in [24] and [14] along with many other pluggable services. Those services can easily be integrated in this architecture. Moreover ACE allows the use of design patterns [27] that can be effectively used to implement multi-threaded models and design-specific servers. For example, our implementation of d-VCD uses the Acceptor-Connector design pattern from the ACE library. The multiple models shown being executed on SystemC-V are a consequence of leveraging the ACE threading mechanisms to simulate concurrent models. The Usage model in Section VII steps through showing how CARH is used. Obviously, there

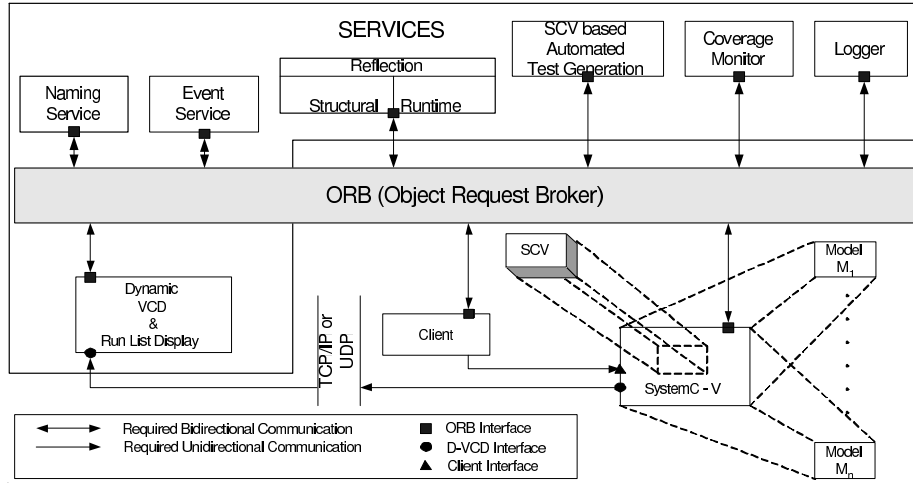


Fig. 1. CARH Software Architecture

is a simulation performance price to pay for using such an infrastructure. Our early experiments show them in tolerable ranges as reported in Section VIII.

VI. SERVICES RENDERED BY CARH

The ability to introspect structural characteristics of a model promotes a large variety of services, of which we currently implement the automated testbench generator and logger services. However, with emphasis on improving methodologies for debugging and model building experience, we extract runtime characteristics from the model as well and describe the d-VCD service. These two services made possible by the introspective architecture are not to be thought of as the only possible services, but simply two of the many that may follow. In addition, with the inherent facility of CORBA with the inclusion of TAO, we provide an elegant infrastructure for a distributed test environment. In this section, we describe these services in effort to prescribe examples which can employ the R-I and TAO capabilities.

A. Reflection Service

Doxygen, XML & Structural reflection: Processing C++/SystemC code through Doxygen to yield its XML documentation is ideal for data reflection purposes. The immediate advantages are that Doxygen output inserts XML tags where it recognizes constructs specific to the language such as C++ classes and it also preserves the source code line by line. The first advantage makes it easy to target particular class objects for further extraction and the latter allows for a well-defined medium for extracting any information from the source that may not be specific to an implementation. Since all SystemC constructs are not recognized during this pre-processing, we use the well-formed XML format of the source code as input to an XML parser to extract further structural information.

We leverage this well-formed XML-based Doxygen output to extract all necessary SystemC constructs not tagged by Doxygen itself using the Xerces parser and we implemented an additional C++ library to generate an Abstract System

Level Description (ASLD). ASLD is written in a well-formed XML format that completely describes each module with the following information: signal and port names, signal and port types and bitwidths, embedded SystemC processes and their entry functions, sensitivity list parameters, and its hierarchy. We introduce a DTD specific for SystemC constructs that validates the ASLD for correctness of the extraction. The reflection service reads the ASLD and populates a data structure representing the model. Finally, TAO/CORBA client and server interfaces are written to make the reflection into a floating CORBA facility accessible via the interfaces. Our website [28] contains all the details and code.

Given an overview of our introspective architecture, we continue to present details on the infrastructure for introspection, with SystemC being the targeted SLDL of choice. We only provide small code snippets to present our approach and the concept of using Doxygen, XML, Xerces-C++, and C++ data structure to complete the reflection service. We present details of the Doxygen pre-processing, XML parsers employed in extracting uninterpreted information from SystemC source files, our method of storing the reflected meta-data and our data structure allowing for introspection.

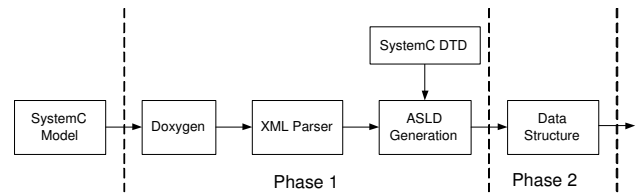


Fig. 2. Design Flow for Reflection Service

Doxygen pre-processing: Using Doxygen has the benefit of simplifying C/C++ parsing and its corresponding XML representations. However, Doxygen requires declaration of all classes for them to be recognized. Since all SystemC constructs are either, global functions, classes, or macros, it is necessary to direct Doxygen to their declarations. For example, when Doxygen executes on just the SystemC model

then declarations such as `sc_in` are not tagged, since it has no knowledge of the class `sc_in`. The immediate alternative is to process the entire SystemC source along with the model, but this is very inconvenient when only interested in reflecting characteristics of the SystemC model. However, Doxygen does not perform complete C/C++ compilation and grammar check and thus, it can potentially document incorrect C/C++ programs. We leverage this by adding the class definition in a file that is included during pre-processing and thus indicating the classes that need to be tagged. There are only a limited number of classes that are of interest and they can easily be declared so Doxygen recognizes them. As an example we describe how we enable Doxygen to tag the `sc_in`, `sc_out`, `sc_int` and `sc_uint` declarations. We include this description file every time we perform our pre-processing such that Doxygen recognizes the declared ports and datatypes as classes. A segment of the file is shown in Figure 3, which shows declaration for input and output ports along with SystemC integer and SystemC unsigned integer datatypes.

```

/*! SystemC port classes !*/
template<class T> class sc_in { };
template<class T> class sc_out { };

/*! SystemC datatype classes !*/
template<class T> class sc_int { };
template<class T> class sc_uint { };

```

Fig. 3. Examples of class declarations

The resulting XML for one code line is shown in Figure 4. Doxygen itself also has some limitations though; it cannot completely tag all the constructs of SystemC without explicitly altering the source code, which we avoid doing. For example, the `SC_MODULE(arg)` macro defines a class specified by the argument `arg`. Since we do not include all SystemC files in the processing, Doxygen does not recognize this macro when we want it to recognize it as a class declaration for class `arg`. However, Doxygen allows for macro expansions during pre-processing. Hence, we insert a pre-processor macro as: `SC_MODULE(arg)=class arg: public sc_module` that allows Doxygen to recognize `arg` as a class derived from class `sc_module`. We define the pre-processor macro expansions in the Doxygen configuration file where the user indicates which files describe the SystemC model, where the XML output should be saved, what macros need to be run, etc. We provide a configuration file with the pre-processor macros defined such that the user only has to point to the directory with the SystemC model. More information regarding the Doxygen configuration is available at [23].

Even through macro pre-processing and class declarations, some SystemC constructs are not recognized without the original SystemC source code. However, the well-formed XML output allows us to use XML parsers to extract the untagged information. We employ Xerces-C++ XML parsers to parse the Doxygen XML output, but we do not present the source code here as it is simply a programming exercise, and point the readers at [29] for the source code.

XML Parsers: Using Doxygen and an XML parser we

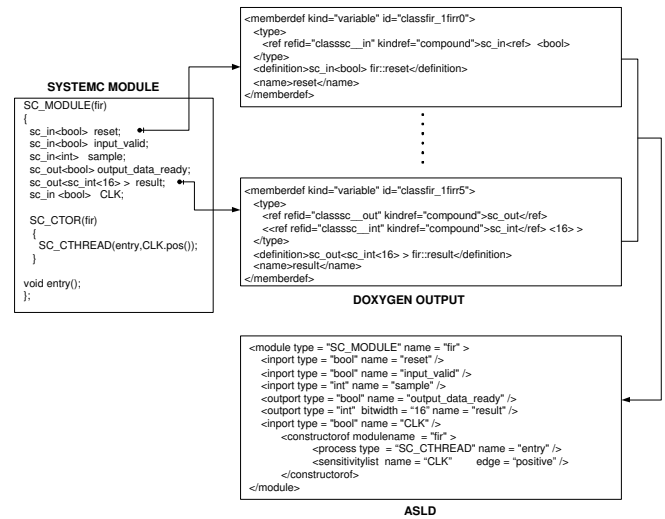


Fig. 4. Doxygen XML Representation for `sc_in`

reflect the following structural characteristics of the SystemC model: port names, signal names, types and widths, module names and processes in modules and their entry functions. We reflect the sensitivity list of each module and the netlist describing the connections including structural hierarchy of the model stored in the ASLD. This ASLD validates against a Document Type Definition (DTD) which defines the legal building blocks of the ASLD that represents the structural information of a SystemC model. Some constraints the DTD enforces are that two ports of a module should have distinct names and all modules within a model should be unique. All these constraints ensure that the ASLD correctly represents an executable SystemC model. The main entities of the ASLD are shown in Listing 1.

ASLD: In Listing 1, the topmost *model* element corresponds to a SystemC model with multiple modules. Each *module* element acts as a container for the following: input ports, output ports, inout ports, signals and submodules. Each *submodule* in a *module* element is the instantiation of a module within another module. This way the ASLD embeds the structural hierarchy in the SystemC model and allows the introspective architecture to infer the toplevel module. The *submodule* is defined similar to a *module* with an additional attribute that is the instance name of the submodule. The *signal* element with its name, type and bitwidth attributes represents a signal in a module. Preserving hierarchy information is very important for correct structural representation. The element *inport* represents an input port for a module with respect to its type, bit width and name. Entities *outport* and *inoutport* represent the output and input-output port of a module. Line 16 describes the *constructorof* element, which contain multiple process elements and keeps a *sensitivitylist* element. The *process* element defines the entry function of a module by identifying whether it is an `sc_method`, `sc_thread` or `sc_thead`. The *sensitivitylist* element registers each signal or port and the edge that a module is sensitive to as a *trigger* element. Connections between submodules can be found either in a module or in the `sc.main`. Each connection element holds the

name of the local signal, the name of the connected instance and the connected port within that instance. This is similar to how the information is present in the SystemC source code and is sufficient to infer the netlist for the internal data structure.

Using our well-defined ASLD, any SystemC model can be translated into an XML based representation and furthermore models designed in other HDLs such as VHDL or Verilog can be translated to represent synonymous SystemC models by mapping them to the ASLD. This offers the advantage that given a translation scheme from say a Verilog design to the ASLD, we can introspect information about the Verilog model as well.

Listing 1. Main Entities of the DTD

```

1<!ELEMENT model (module)* >
2<!ATTLIST model name CDATA #REQUIRED>
3
4<!ELEMENT module (inport | output | inoutport |
5  signal | submodule)* >
6<!ATTLIST module name CDATA #REQUIRED type CDATA #
7  REQUIRED >
8
9<!ELEMENT submodule EMPTY >
10<!ATTLIST submodule type CDATA #REQUIRED name CDATA #
11  REQUIRED instance CDATA #REQUIRED >
12
13<!ELEMENT signal EMPTY >
14<!ATTLIST signal type CDATA #REQUIRED bitwidth CDATA
15  #IMPLIED name CDATA #REQUIRED >
16
17<!ELEMENT inport EMPTY >
18<!ATTLIST inport type CDATA #REQUIRED bitwidth CDATA
19  #IMPLIED name CDATA #REQUIRED >
20
21<!ELEMENT constructorof (process * | sensitivitylist)
22  >
23<!ATTLIST constructorof modulename CDATA #REQUIRED >
24
25<!ELEMENT process EMPTY >
26<!ATTLIST process type CDATA #REQUIRED name CDATA #
27  REQUIRED >
28
29<!ELEMENT sensitivitylist (trigger)* >
30
31<!ELEMENT trigger EMPTY >
32<!ATTLIST trigger name CDATA #REQUIRED edge CDATA #
33  REQUIRED>
34
35<!ELEMENT connection EMPTY>
36<!ATTLIST connection instance CDATA #REQUIRED member
37  CDATA #REQUIRED local_signal CDATA #REQUIRED>

```

Data structure: The ASLD serves as an information base for our introspection capabilities. We create an internal data structure that reads in this information, enhances it and makes it easily accessible. The class diagram in Figure 5 gives an overview of the data structure. The *topmodule* represents the toplevel module from where we can navigate through the whole application. It holds a list of module instances and a list of connections. Each connection has one read port and one or more write ports. The whole data structure is modeled quite close to the actual structure of SystemC source code. All information about ports and signals and connections are in the module structure and only replicated once. Each time a module is instantiated a *moduleinstance* is created that holds a pointer to its corresponding module.

The information present in the ASLD and the data structure does not contain any behavioral details about the SystemC model at this time, it merely gives a control perspective of the system. It makes any control flow analysis and optimizations on the underlying SystemC very accessible.

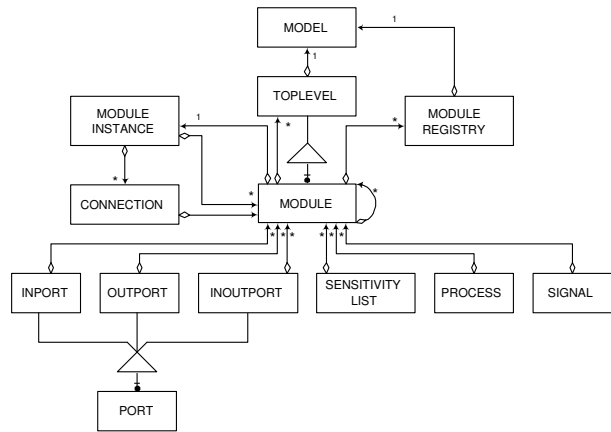


Fig. 5. Class diagram showing data structure

Focus-defocusing of models and modules: The reflection service also implements the idea of *focusing* on modules. Since a model generally consists of multiple SystemC modules, the reflection service reflects information regarding the module in *focus*. This *focus* can be changed through interface functions accessible to the user. Furthermore, there can be multiple instances of the same or different model that needs reflecting. Similar to *focusing* on modules, we implement *focus* support for models.

The source code for this introspective infrastructure in SystemC is available online at [29] for download.

B. Testbench Generator

CARH provides automated test generation and coverage monitoring as vertical services for testing SystemC models. The test generation service is built using the SystemC Verification (SCV) [1], which is a library of C++ classes, that provide tightly integrated verification capabilities within SystemC. This service takes the name of the model, a data file and a few user specified parameters to generate customized testbenches for the model.

Algorithm 1 Test Generation

```

1: Invoke generate_moduletb(module m, option p, datafile d)
2: if marked ports exist then
3:   For each marked port 'pt'
4:     if 'pt' exists in module 'm' then
5:       if p = "unconstRand" then
6:         Query reflection service for type of port 'pt'
7:         Query reflection service for bitwidth of port 'pt'
8:         Generate randomized testbench
9:       else if p = "simpleRand" then
10:        Repeat Step 2 & 3
11:        Generate simple constrained randomized testbench
12:       else if p = "distRand" then
13:        Repeat Step 2 & 3
14:        Generate randomized testbench with distribution modes
15:       end if
16:     else
17:       Print "Test Generation Failed"
18:     endif
19:   else
20:     For all ports in module 'm'
21:       Repeat through Steps 4 to 18
22:     end if

```

The client issues commands that specify the model and module in *focus* to the test generation service, which invokes

the respective API call on the reflection object that creates the corresponding ASLD for the SystemC model. Then it invokes the API call for initialization of the data structure and enabling the introspective capabilities of the reflection object. This allows the test generator to introspect the reflected information and automatically generate a testbench based on the user-defined data file and parameters. The test generator searches for marked ports for the given module and introspects them. If none were found, then all the ports for the given module are introspected and a corresponding testbench is generated.

The test generator can create constrained and unconstrained randomized testbenches. In the *unconstRand mode*, unconstrained randomized testbenches are created, which use objects of *scv_smart_ptr<T>* type from SCV. This is the default mode of the test generator. In the *simpleRand mode*, constrained randomized testbenches are created. These testbenches issue *Keep_out* and *Keep_only* commands to define the legal range of values given by the user in the data file. Similarly in the *distRand mode*, *scv_bag* objects are used in testbenches given the appropriate commands from the console and providing the data file with the values and their probability. Figure 6 and 9 show snippets of executing the FIR example using CARH.

1) *Testbench generation Example*: We briefly describe the testbenches generated using the FIR example from SystemC distribution. In particular, we set focus on the computation block of the FIR. We present Figure 6 that shows three testbenches using the *unconstRand*, *simpleRand* and *distRand* modes. The *unconstRand* generates unconstrained randomized testbenches, the *simpleRand* constrain the randomization using *keep_out* and *keep_only* constructs with legal ranges specified from an input data file and the *distRand* defines *SCV_bags* that give a probabilistic distribution for the randomization. Once, the automated testbench generated, it is integrated and compiled to test the FIR block. The integration is performed manually by defining the appropriate interface between the generated testbench and the FIR block.

<pre> #!/ Defining an SCV smart pointer !/ scv_smart_ptr <int> r_sample; #!/ Generating the randomized values !/ r_sample->next(); </pre> <p>Snippet of the testbench in the <i>unconstRand</i> mode</p>	<pre> #!/ Defining weights for the distribution mode !/ scv_smart_ptr <int> r_sample; scv_bag<pair<int,int>> d_sample; #!/ Defining the legal ranges !/ d_sample.add(pair<int,int> (1, 3), 40); d_sample.add(pair<int,int> (5, 7), 30); #!/ Setting the distribution mode !/ r_sample->set_mode(d_sample); </pre> <p>Snippet of the testbench in the <i>distRand</i> mode</p>
<pre> #!/ Defining simple constraints !/ scv_smart_ptr <int> r_sample; #!/ Defining the legal ranges !/ r_sample->keep_only (10,100); r_sample->keep_out (21, 49); r_sample->keep_out (61, 89); </pre> <p>Snippet of the testbench in the <i>simpleRand</i> mode</p>	

Fig. 6. Code snippets for generated testbenches

We intend to improve our automated testbench generation capabilities by first implementing additional services such as coverage monitors and simulation performance monitors to better analyze the SystemC model. These additional services will assist the testbench generator in making more intelligent and concentrated testbenches.

Distributed Test Environment: Currently, the user compiles

the testbench with the model through the console. However, we target CARH to handle compilation and execution as described in [14] and shown in Figure 7. This allows the testbenches to compile and execute independently through interface calls on the ORB. In effect, this distributed test environment also supports cosimulation where different languages that support CORBA interfaces may communicate through the ORB. Testbenches could be written in languages different from regular HDLs as long as they interface correctly with the ORB. Furthermore, visualization tools interfacing through the ORB can greatly benefit from such an architecture. Pseudocode 1 shows steps involved in generating a testbench for a module and the interaction between the test generation and reflection service.

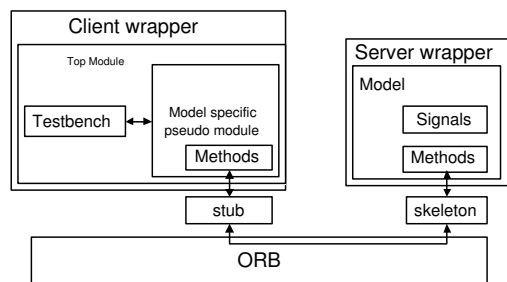


Fig. 7. Test Environment

C. d-VCD Service

Implementing runtime reflection mandates alterations to the existing SystemC source. This is unavoidable if runtime information has to be exposed for SystemC and we justify this change by having two versions of SystemC. We call our altered version SystemC-V, which the designer can use for the purpose of verification and debugging of SystemC models. However, for fast simulation the same model can be compiled with the original unaltered version of SystemC by simply altering the library target in the Makefiles.

The d-VCD service displays signal value changes for a module “as they happen”. Regular VCD viewers display VCD information from a file generated by the simulation. However, we enable the d-VCD viewer to update itself as the signals of a focused module in the SystemC model changes. Every signal value change for the module in focus communicates with the d-VCD. Likewise, at every delta cycle we send the process names on the runlist to the d-VCD. Figure 8 shows a screenshot of a GUI for the VCD using Qt [30]. To enable SystemC-V to expose this information we altered the *sc_signal* class along with adding an extra class. Before discussing brief implementation details it is necessary to understand how we utilize the reflection service. In order to gain access to the reflected information, we instantiate an object of class *module* and use it as our introspective mechanism. Member functions are invoked on the instance of *module* to set the focus on the appropriate module and to introspect the characteristics of the module.

To facilitate SystemC for exposing runtime characteristics, we implement class *fas_sc_signal_info* that stores the

signal name (`sig_name`), signal type (`sig_type`) and classification type (`sig_class`) with their respective set and get member functions. SystemC has three class representations for `sc_signal`, where the first one is of template type `T`, the second is of type `bool` and the third is of type `sc_logic`. Each of these classes inherit the `fas_sc_signal_info` class minimizing changes to the original source. In fact, the only changes in the original source are in the constructor and the `update()` member functions. We require the user to use the explicit constructor of the `sc_signal` class such that the name of the signal variable is the same as the parameter specified in the constructor. This is necessary such that an object of `sc_signal` concurs with the introspected information from the reflection service. We also provide a PERL script that automatically does this. The `update()` function is responsible for generating SystemC events when there is a change in the signal value and an ideal place to transmit the data to the d-VCD service. So, if the signal type is a SystemC type then the `to_string()` converts to a `string` but if it is classified as a C++ type then it is converted using `stringstream` conversions.

The explicit constructors invoke `classify_type()` which classifies the signal into either a SystemC type or a C++ type. We use the classification to convert all C++ and SystemC types values to a `string` type. This is such that multiple VCD viewers can easily interface with the values returned from SystemC-V and they need not be aware of language specific datatypes. Since all SystemC datatypes have a `to_string()` member function, it is easy to return the string equivalent for the value. However, for C++ datatypes we employ a work around using `stringstream` conversion to return the string equivalent. Even though, we are successfully able to translate any native C++ and SystemC datatypes to their string equivalent, the compilation fails when a SystemC model uses signals of C++ and SystemC types together. This is because for C++ datatypes the compiler cannot locate a defined `to_string()` member function. An immediate solution to this involves implementing a templated container class for the templated variables in `sc_signal` class such that the container class has a defined function `to_string()` that allows correct compilation. We add the class `containerT<T>` as a container class and replace variable instances of type `T` to `containerT<T>` in order to circumvent the compilation problem. We interface the runtime VCD with the Qt VCD viewer implemented by us, shown in Figure 8.

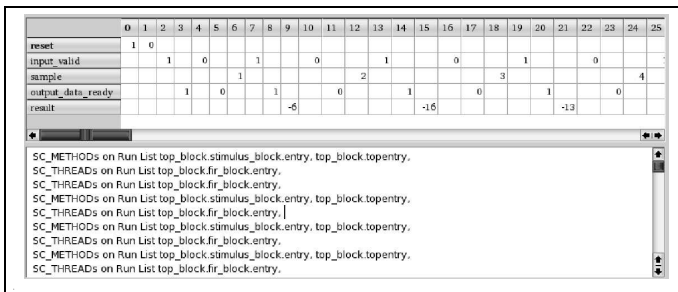


Fig. 8. d-VCD Output

With dynamic runtime support in SystemC, we also expose runlist information as another example of runtime reflection. Being able to see the processes on the process runlists gives a significant advantage during debugging. This capability is available by altering the SystemC `sc_simcontext` class. Figure 8 also shows the output for the processes on the runlist along with the dynamic value changes on signals. Exposing the process name to the d-VCD service itself does not require the reflection service since the process names are available in SystemC `sc_module` class via the `name()` member function. However, using the reflection service we provide the user with more concentrated visuals of model execution by enabling the user to specify which particular module's processes should be displayed. This requires querying the reflection service for the name of the modules in focus and only returning the names of the processes that are contained within those modules. Implementing this capability required stepping through SystemC runlist queues and monitoring whether they match the modules of interest and transmitting the name to the d-VCD service.

VII. USAGE MODEL OF CARH

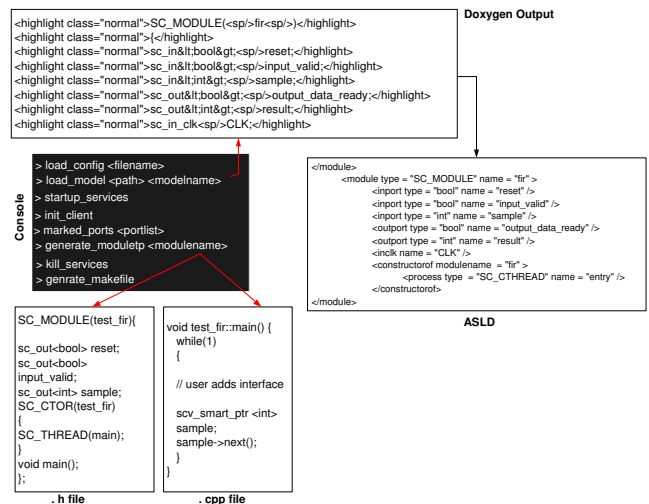


Fig. 9. Snippets of Intermediate files

User Console: We developed an interactive shell for CARH, which is shown as Client in Figure 1. The user initiates the shell and interacts with it to setup the services, request for a testbench, and execute testbench and model. Figure 10 shows an interaction diagram with the complete usage model. The usage model starts at the point where the user needs verification of a SystemC model and initiates the interactive shell to do so as shown in Figure 9.

Load configuration: The first step is to *load the configuration file* using the `load_config <path>` command. This loads the system specific paths to the required libraries and executables.

Specify model: Assuming that the configuration paths are correct, the user invokes the `load_model <path> <modelname>` command to specify the directory of the model under investigation and a unique `modelname` associating it.

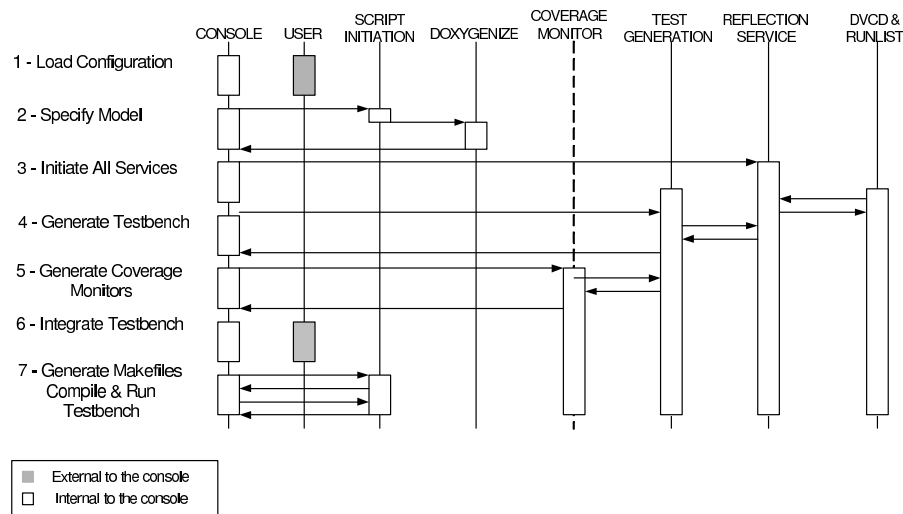


Fig. 10. CARH's Interaction Diagram

```

SYSTEMC=/home/deepak/sc-ace-rdy-2.0.1
ACE_ROOT=/home/deepak/ace/ACE_wrappers
TAO_ROOT=/home/deepak/ace/ACE_wrappers/TAO DOXY_ROOT=/usr/bin
REFLECTION_SVC=/home/deepak/ace/ACE_wrappers/Reflection
NAMING_SVC=NameService
TESTGEN_SVC=/home/deepak/ace/ACE_wrappers/Reflection/tgn
DVCD_SVC=/home/deepak/ace/ACE_wrappers/Reflection/dvcd

```

Fig. 11. Snapshot of Configuration file

This command flattens the model into a single flat file by concatenating all *.h and *.cpp files in the specified directory followed by running Doxygen on this file. This results in a well-formed XML representation of the SystemC model. For an example see the FIR model in Figure 9 that shows snippets of the Doxygen output and the ASLD generated.

Initiate services: To start up CARH's services, the user must invoke `startup_services` followed by `init_clients`. These two commands first start the reflection service followed by the d-VCD and then the automated test generator.

Mark ports: Ports can be marked that are given as input to the test generator. The user can also create a file with the listing of the module names and the ports of interest and load these settings. However, the command `marked_ports <portlist>` initializes the ports in `portlist` for introspection and test generation focusing on these ports.

Test generation: The users can then request testbenches by calling `generate_module_tb <module_name>` or `generate_top_level_tb`. We only present the default test generation commands in this usage model. The `generate_module_tb` creates a testbench for the specific `module_name`. However, the user can use `generate_top_level_tb` command to generate a testbench for the entire model, except that this requires wrapping the entire model in a toplevel module. We require the user to wrap it in a module called `systemc_data_introspection` to indicate to the reflection service the toplevel module. The user can also request for the Coverage Monitor service for the appropriate test. Figure 9 also shows the .h and the .cpp file generated for the computation module of the FIR.

Behavioral information: After successful testbench generation, the user needs to add interface information into the test such that it can be integrated with the model. Consider the computation module of the FIR, the interfacing information would be regarding how many cycles the reset needs to be applied and when the inputs need to be sampled. Such behavioral aspects are not yet automated with our test generation service. Then, the user can use the `generate_makefile` command to produce *Makefiles* specific for the model after which the testbench can be compiled and executed from the console. While the model executes, the d-VCD service does a dump of the different value change that occur across the various signals of the model. A screenshot of the d-VCD for the FIR example is shown in Figure 8. It also displays the processes on the runlist.

VIII. SIMULATION RESULTS

Samples	FIR (seconds)		FFT (seconds)	
	Original	CARH	Original	CARH
100000	4.16	189.42	2.91	191.17
200000	8.20	384.02	5.81	377.47
300000	12.41	566.25	8.79	586.58
400000	16.54	757.02	11.70	788.60
500000	20.67	943.23	14.62	984.03

TABLE I
SIMULATION RESULTS ON FIR & FFT

Table I shows simulation times in seconds for two examples: Finite Impulse Response (FIR) filter and Fast Fourier Transform models. Columns marked *Original* refers to the model with the testbench generated from the test generation service compiled with SystemC 2.0.1 and SCV version 1.0p1 and CARH refers to the testbench compiled with SystemC-V and CARH infrastructure. There is a performance degradation of approximately 45x and 65x for the FIR and FFT models respectively. This performance decrease is not surprising when using TAO. There is a significant overhead in communicating through an ORB as each of the services are spawned as

separate processes registered with the ORB that require communication through CORBA interfaces. However, the facilities provided by CARH justifies the performance degradation because CARH offers multiple services and possibilities for various extensions. CARH can be used during development of the model for debugging, testing and so on, and when a simulation needs to be faster then the model can be easily linked to original SystemC and SCV libraries since no changes in the source are made. In fact, since services can be turned on and off at the designers will, we employed the R-I and testbench generation service to create our testbenches for the FIR and FFT examples, after which we simply compiled and executed it with the unaltered version of SystemC. The simulation times were on average the same as the Original timings shown in Table I.

Another important point to note is that TAO is just one implementation of CORBA and not the best stripped down version suitable for CARH. For example, it has real-time capabilities that we do not employ at all. We only use TAO as a solution to display our proof of concept for service orientation and industrial strength solutions could use a much lighter and smaller CORBA-variant. This would impact the simulation results directly as well.

IX. OUR EXPERIENCE WITH CARH

The foremost important experience we discuss here involves adding a service to CARH. We recount briefly the steps in integrating this service without getting into the details of the algorithms employed. We also briefly describe our debugging experience with the d-VCD and R-I services. This is followed by a table that describes some of the features CARH possess versus existing commercial tools.

So, for example, determining the number of times each SystemC process triggers could be a coverage metric sought after. Evidently, this requires runtime reflection that the R-I service provides. We implement the coverage service in C++ (or any CORBA-compliant language) and begin by constructing a data structure that can store the SystemC process names and an integer value representing the trigger count. We define the appropriate member functions that can be used to populate this data structure. This first step of implementing the data structure and its API in C++ is a programmer dependent task regarding the time required to implement. For us, it took us less than an hour. Uptil this stage we are only preparing the coverage monitor implementation and the following step involves wrapping this C++ implementation of the coverage monitor with CORBA service methods. However, before this, the IDL must be defined through which this coverage monitor service interacts with the other clients and services across the ORB. This step is almost similar to specifying the API of the data structure, since most of the member functions of the coverage monitor should be accessible via the ORB. Hence, the IDL construction itself requires little time. The third stage involves instantiating an ORB through which the service registers itself with the Naming Service allowing other services to locate this new service. We provide a simple wrapper class that allows easy integration of any generic C++

Capability	SystemC Studio	ConvergenceSC	Incisive	CARH
Extendibility	No	No	No	Yes
Interoperability	No	No	No	Yes
Services				
Reflection	No	No	No	Yes
Introspection	No	No	No	Yes
Test Generation	Yes	No	Yes	Yes
Coverage Monitors	Yes	No	Yes	No
Debugging (Call graphs, Execution traces, etc.)	No	Yes	Yes	Yes

TABLE II
BRIEF COMPARISON BETWEEN COMMERCIAL TOOLS AND CARH

implementation into a service making this third step requiring only minutes. Finally, the implementation for requesting information from the R-I service through the ORB and populating the data structure is added to the coverage monitor. This completes the integration of our example of a basic monitor service into CARH. The entire process takes only a few hours, depending on the complexity of the actual core of the service. For services that require more complex data structures, algorithms etc., the most of the integration time is spent in programming these algorithms and not the integration. Furthermore, if there are existing libraries of algorithms, they can be easily used to provide the core technology / algorithms requiring the programmer to only write the IDL and wrapping the implementation into a service. The services we experiment with are mainly implemented by ourselves. We do not integrate a commercial tool into CARH as of yet.

Our debugging experience is limited to using the R-I and d-VCD services. We found that these two services did indeed help us reduce the debugging time, especially the display of processes being triggered. One interesting problem we discovered with our model using these services is a common one of missed immediate notification of an `sc_event`. In SystemC 2.0.1, there is no notion of an event queue for `sc_events` added into the model for synchronization purposes. This is rectified in SystemC 2.1 with the introduction of an event queue such that no notifications are missed, but instead queued. However, with SystemC 2.0.1 (the version we use for our development) we were able to pinpoint and freeze the simulation just before the notification and then just after, to see which processes were triggered following the immediate notification. However, this is simply one experience of locating one bug. We understand that adding other debugging facilities will greatly improve locating design errors.

We present a table displaying some of the noticeable features in commercial tools compared with CARH. We base our comparison only on publicly available information and do not claim to know the details of the underlying technologies used in these commercial tools. From Table II shows some of the feature comparisons.

X. CONCLUSION

CARH presents a methodology where we employ the use of public-domain tools such as Doxygen, Apache's Xerces-C++, TAO, and ACE to present a service oriented validation architecture for SystemC. In our case, we chose our SLDL to be SystemC, however that only serves as an example. We describe our approach in detail and also present CARH whose core feature is the introspective architecture. Services such as the automated test generator and d-VCD are some of the examples that utilize this R-I capability in SystemC. The use of services and the ORB suggests a tidy deployment strategy for extensions to a framework. Furthermore, using a CORBA-variant needs the extensions to adhere a strict interface that can easily be added as wrappers. This allows the design of the actual feature to remain isolated from the interface and can communicate through messages promoting reuse of technology. Even though there is a performance degradation in simulation results, the exercise of building highly complex models can be eased. For fast simulation, once the models are tested and verified, can be easily linked to the original SystemC libraries. Since, the services can be turned off and on at the designer's will, the simulation times may not be affected at all if say only the testbench generator service is employed. The main overhead comes in when using SystemC-V which requires exposing the internal characteristics of the model under test via an ORB. The simulation experiments showed that having all the services up induces a significant performance penalty. The performance of a simulation run in this framework is not optimized because we want to create a "proof of concept" and hence we used an existing CORBA implementation which is optimized for real-time middleware applications, and hence not necessarily customized for this application. However, we believe that if this idea catches on, we will create customized middleware compliant with OMG CORBA specifications that will be optimized for this specific purpose and hence we will have much better performance. However, to show the validity of such a plug-n-play validation framework and infrastructure, we did not feel the need to demonstrate performance efficiency but rather show the viability of the implementation and illustrate the advantages.

The one most fundamental issue exposed in this work is of service-orientation and using a standardized specification for providing an architecture for validating system level models. With the use of the OMG specification and its implementation of the CORBA-variant TAO, we show the advantage of easy deployment, integration and distribution of features as services. Furthermore, due to the OMG standardization, any implementation abiding by this standardization would be easy to integrate into CARH. The ease of extendibility is natural with such an architecture. Having said that, CARH is a proof of concept for promoting a service oriented architecture and for better performance results, a light-weight CORBA-variant other than TAO should be used. On the other hand, CARH's extendibility can result in numerous useful tools for debugging, visualization, performance monitoring, etc. We simply show some of the possibilities to further promote the benefits of a service oriented architecture.

Our experience with CARH suggests an improved model building experience. In particular, automated testbench generation overcomes the need to create basic sanity test cases to verify the correctness of the design. Even with the basic automated testbench generation service, we could easily locate problematic test cases resolves as bug fixes in our designs. With the addition of more intelligent testbench generation algorithms using information from the coverage analysis service, we foresee a much improved test environment. As for the d-VCD, the process list visualization and the value changes are definitely useful. However, we understand that the need for step-wise execution of the model is crucial and the ability to pause the simulation is needed. We plan to implement this using the push-pull event service of TAO such that designers can step through event notifies and module executions.

Part of this work, the SystemC parser, is available as an open-source project called SystemCXML [29]. We also implement an automated test generation service that uses the existing SCV library to automatically generate testbenches for SystemC models in CARH. In addition to that, we offer dynamic representation of the value changes shown by introducing the d-VCD service along with process list information. We implement a console through which a user controls this entire framework. We briefly present a list to summarize the features that we have implemented so far in CARH. (i) Reflection service provides structural and runtime information. (ii) Test generation service supports constrained and unconstrained randomized testbenches using information from the reflection service. (iii) Naming service used to access all other services on an ORB. (iv) d-VCD service receives signal changes and runlist information. This uses an Acceptor-Connector design pattern from ACE. (v) Client console allows integration of all the services.

REFERENCES

- [1] OSCI, "SystemC and SystemC Verification," Website: <http://www.systemc.org>.
- [2] SPECC, "SpecC," Website: <http://www.ics.uci.edu/specc/>.
- [3] SystemVerilog, "System Verilog," Website: <http://www.systemverilog.org/>.
- [4] Novas, "Debussy Debug System," <http://www.novas.com>.
- [5] CoWare, "ConvergenSC," <http://www.coware.com>.
- [6] Synopsys, "Smart RTL Verification," <http://www.synopsys.com>.
- [7] Cadence, "Incisive Functional Verification," <http://www.cadence.com>.
- [8] TAO, "Real-time CORBA with TAO (The ACE ORB)," <http://www.cs.wustl.edu/~schmidt/TAO.html>.
- [9] OMG, "OMG CORBA," <http://www.corba.org/>.
- [10] ACE, "Adaptive Communication Environment," <http://www.cs.wustl.edu/~schmidt/ACE.html>.
- [11] E. Group, "Edison C++ Front-End," <http://www.edg.com>.
- [12] VHDL, "VHDL," Website: <http://www.vhdl.org/>.
- [13] OMG, "OMG," <http://www.omg.org/>.
- [14] Hamabe, "SystemC over CORBA," <http://www5a.biglobe.ne.jp/~hamabe/index.html>.
- [15] D. Berner, H. Patel, D. Mathaikutty, S. Shukla, and J. Talpin, "SystemCXML: An Extensible SystemC Front End Using XML," Virginia Tech, Tech. Rep., 2005.
- [16] E. R. Gansner and S. C. North, "An open graph visualization system and its applications to software engineering," *Softw. Pract. Exper.*, vol. 30, no. 11, pp. 1203–1233, 2000.
- [17] L. Charest, M. Reid, E. Aboulhamid, and G. Boi, "Introspection in System-Level Language Frameworks: Meta-level vs. Integrated," in *Proceedings of Design and Test Automation in Europe*, 2003.
- [18] W. Snyder, "SystemPerl," <http://www.veripool.com/systemperl.html>.

- [19] F. Doucet, S. Shukla, and R. Gupta, "A Methodology for Interfacing Open Source SystemC with a Third Party Software," in *Proceedings of Design and Test Automation in Europe*, 2001.
- [20] GreenSocs, "Pinapa: A SystemC Frontend," <http://www.greensocs.com/>.
- [21] J. Lapalme, E. M. Aboulhamid, G. Nicolescu, L. Charest, F. R. Boyer, J. P. David, and G. Bois, ".NET Framework – A Solution for the Next Generation Tools for System-Level Modeling and Simulation," in *Proceedings of Design and Test Automation in Europe*, 2003.
- [22] Mono, "Mono Project," <http://www.mono-project.com/>.
- [23] Doxygen Team, "Doxygen," <http://www.stack.nl/~dimitri/doxygen/>.
- [24] A. Amar, P. Boulet, J. Dekeyser, S. Meftali, S. Niar, M. Samyn, and J. Vennin., "SoC Simulation," <http://www.inria.fr/rapportsactivite/RA2003/dart2003/>.
- [25] L. Formaggio and G. P. F. Fummi, "A Timing-Accurate HW/SW Co-simulation of an ISS with SystemC," in *Proceedings of International Conference on Hardware/Software Codesign and System Synthesis*, 2004.
- [26] NS2, "The Network Simulator - NS-2," <http://www.isi.edu/nsnam/ns/>.
- [27] E. Gamma, R. Helm, J. R. and J. Vlissides, *Design Patterns*. Addison Wesley, 1995.
- [28] The FERMAT Group, "Formal Engineering REsearch with Models, Abstractions and Transformations," Website: <http://fermat.ece.vt.edu>.
- [29] D. A. Mathaikutty, D. Berner, H. D. Patel, and S. K. Shukla, "FERMAT's SystemC Parser," <http://systemcxml.sourceforge.net>, 2004.
- [30] Troll Tech, "Qt," Website: <http://troll.no>.

Details of the revision made to this version of the paper

<**Comment from the Associate Editor**> Please address all of the issues raised by reviewer 1, as well as suggestions, and write a response to that review. I'll then make a decision – I don't think another full review cycle will be needed, unless I feel that the issues were not thoroughly addressed.

<**Our Comment**> We have made the recommended alterations to this version. We added a substantial section describing our experience using CARH in Section IX. In particular, we first discuss the steps involved in integrating a coverage monitor service to CARH. Here, we take a simple example of counting the frequency of triggered SystemC processes as a coverage metric that employs the R-I service and indicate that the main engineering effort is in the implementing the core technology instead of the actual integration into CARH. We follow this by a brief discussion on our debugging experience using the d-VCD and R-I service. Again, we employ a small but common problem with SystemC 2.0.1 of missed notifications. Finally, we add a table comparing commercial tool features and CARH to the best of our knowledge available from public domain sources. Unfortunately, we don't have the resources to explore commercial tools for integration with CARH and we believe it would not be possible to obtain source code for these tools to understand the interface requirements either.

We have also added a main contributions section to highlight the fundamental contributions in this work. We also pinpoint the one most important issue of service-orientation that is necessary for easy deployment, integration and extendibility. Furthermore, any implementation of the OMG standard would allow integration into CARH.

We hope that these additions clarify the issues and we would like to thank the reviewers for improving the quality of this contribution with their continual extensive critiques and suggestions.

Addressing Reviewer 1's concerns

<**Reviewer 1**> The issues raised in my review (and other's reviews) have been properly addressed. The focus of the paper (including title) has been properly shifted towards the actual proof-of-concept implementation of an extensible design framework using public- domain tools. In other words, the work is honestly described and well-written.

Towards the contribution of the work, I am still missing convincing support of the benefits of the approach. What is novel? Or, what is better than other approaches? These questions need to be answered by providing experimental evidence and results that allow a comparison.

<**Our Comment**> We hope that the novelty is more apparent after adding the main contributions. With the multitude of commercial tools out there which prohibit cross-integration or any integration for that matter, it makes it very difficult for any designer to be satisfied with just one tool, or even enhance the commercial tool. The idea is that if these commercial tools followed a service-orientation approach following a standardization, then their object code could still be proprietary but allows designers to add, enhance and append to the existing commercial tool. Other commercial tools could leverage the interface defined by them to allow cross-integration of multiple tools.

The only experimental evidence given (simulation times) is very contra-productive as the performance penalty is simply too high. It is understood that performance is a price for the additional services, but in this trade-off, the value of the services must be demonstrated.

<**Reviewer 1**> The only experimental evidence given (simulation times) is very contra-productive as the performance penalty is simply too high. It is understood that performance is a price for the additional services, but in this trade-off, the value of the services must be demonstrated.

Examples of experimental values could be:

- How long does it take to extend the system for another service? (low number \Rightarrow easy extensibility)
- Cost of the system (public domain vs. commercial \Leftrightarrow 0 vs. \$\$\$)
- How much does debugging time decrease by use of introspection/ reflection?
- Development time of an actual system being developed
- Table of features comparing your system against existing ones

The paper contains a lot of implementation details (i.e. the software architecture and its interfaces) and is valuable as such, but is lacking actual insights gained that could apply to other, similar approaches. Thus, the contribution is quite limited.

<**Our Comment**> We have added an entire experience section that discusses these issues. Please see section IX and the response to the associate editor.

Addressing Reviewer 3's concerns

<**Reviewer 3**> By restructuring the paper and changing its focus the authors have addressed my primary concerns with their previous submission. The only remaining comment that I have is that the paper would be improved by more explicitly identifying the fundamental issues that arise in applying the SOA approach rather than requiring the reader to pull specific lessons out of the examples.

<**Our Comment**> We have added a main contributions section that better highlights the important contributions of this work. The fundamental issue in applying SOA are discussed in section IIIA that describe the advantages of service-orientation.

The first and most natural is of extendability since services can be incorporated into the architecture without much disturbance to existing services. The second is the advantage of cross-platform support. An implementation in any language can be easily integrated as long as the interface mechanism conforms to CARH's OMG standard. The immediate advantage of having the OMG standard is an excellent advantage for other tools to follow to provide the above advantages. Lastly, the distributed nature of CORBA allows for these services to be deployed on different locations, computers etc.