Structured Component Composition Frameworks for Embedded System Design

Sandeep K. Shukla[1], Frederic Doucet[2], and Rajesh K. Gupta[2]

[1] Electrical and Computer Engineering Department, Virginia Tech, Blacksburg, VA 24061
[2] Information and Computer Science Department, University of California, Irvine, CA 92697 *

Abstract. The increasing integration of system-chips is leading to a widening gap in the size and complexity of the chip-level design and the design capabilities. A number of advances in high-level modeling and validation have been proposed over the past decade in an attempt to bridge the gap in design productivity. Prominent among these are advances in *Abstraction* and *Reuse* and structured design methods such as Component-Based Design and Platform-Based Design. In this paper, we present an overview of the recent advances in reuse, abstraction, and component frameworks. We describe a compositional approach to high-level modeling as implemented in the BALBOA project.

1 Introduction

Due to advances in microelectronic processing and devices, while fabricating millions of transistors on chip has become easier, the functionalities implemented using these devices have steadily been growing, outstripping manual design capabilities and the capacity of design automation tools. A number of strategies are being explored by the microelectronic designers in an attempt to improve the design productivity and the quality of designs through advances in modeling and validation techniques. Raising the abstraction level at which designs are entered and validated has a direct impact on the design quality and design time. Consequently, much of the recent effort in the area has been focused on the specification methodologies and languages for modeling designs at the system level. The focus of this paper is on component composition frameworks, which provide support for correct composition of existing components and automatic or semi-automatic means for validation checks. We show the techniques that are used to improve component reuse and design productivity using an example framework currently under development.

2 Components and Virtual Components for SOCs

A system-on-chip or SOC refers to a complete system from an end application point of view. In other words, a SOC represents implementation of a complete application on a single chip consisting of a range of building blocks from processors, memory, to communication and networking elements. There may be a bottom-up or a top-down

* This reasearch was partially funded by SRC, FCAR, NSF, CAL-MICRO, CORE

approach to building an application into an SOC. In a bottom up approach, various functionalities of the system are mapped to existing components, and then the components are connected together in such a way that the resulting system has the required functionalities and performance characteristics. In a top-down approach, a functional specification of the application is refined to obtain architectural specification, and synthesized into implementation either by automated synthesis tools (probably in the future), or by manual refinements to synthesizable descriptions.

In the bottom-up approach the application functionality can often be structured into various hardware and software components which can provide parts of the functionality, and as a whole meet the functional and performance goals. Top-down approaches could also yield to refinements that are then mapped to various hardware/software components. So, from that perspective, a component may then be a piece of functionality implemented in software or as a dedicated piece of silicon hardware or a combination of the two. A component may be virtual in that it represents a well-defined functionality without an associated hardware/hardware implementation. The phrase "virtual component" is used to describe reusable IP components which are composed with other components (as if plugging them onto virtual sockets), similar to real hardware components are plugged into real sockets on a board [22]. A virtual component may be turned into a concrete implementation by instantiating and appropriately combining with other components. To quote from the VSIA's statement on the purpose of standardization effort for virtual socket interfaces [22]

> *One solution for this dilemma (productivity gap) is to design with pre-designed blocks, much as is now done using off-the-shelf IC's on printed circuit boards. The pre-designed blocks are a form of Intellectual Property (IP), which is variously referred to as IP, IP blocks, cores, system-level blocks (SLB), macros, system level macros (SLMs), or Virtual Components (VCs – the VSIA term for these elements).*

A virtual component is then defined to be a reusable piece of functionality, that is, its functions may be reused in other applications. However, not every component can be reused across all applications. Reusability requires not only matching of functionality, but an ability to compose the component functionalities in a way that correctly implements the end application. This requires specific component capabilities that we shall discuss later in the context of composition frameworks (tools and methods) that enable SOC application development.

2.1 Component Composition Frameworks

A composition framework provides reasoning capabilities and tools that enable a system designer to compose components into a specific application. These capabilities include selection of the correct components, automated creation of correct interfaces, simulation of the composed design, testing and validation for behavioral correctness and equivalence checks. A limited form of Component composition is common in purely software systems where environments often known as Integrated Development Environments or IDEs are used to facilitate component selection and composition. Compared

to software IDEs such as Microsoft Visual studio, hardware component composition frameworks are more difficult to build. Part of the complexity is due to the various ways in which the integrated circuit blocks are represented, designed and composed. Due to a common model of the execution machine and commonly accepted compiler conventions, software reusable components can be composed fairly easily either statically during compilation or even during runtime. Even with this ease in compilation and runtime library linking, correct application functionality remains a challenge. In the absence of such compiler and middleware conventions, ensuring component composition for silicon hardware is a challenge despite well-understood physics and logic of interface circuits. At higher abstraction levels, often a connection between components is created through limited set of ports and signals in, what is often known as, *structural design for SOCs*. Such a composition presupposes a structural representation for the components. Even if a component is not structural, but behavioral, it can often be composed using special components (e.g., protocol modules) interconnecting the components. Further, one often is faced with composing components described at different levels of abstraction, one component at behavioral level (e.g., as an algorithm) and another in a register transfer description. Composability of models can be defined along a number of modeling dimensions [7]:

1. *Temporal detail*: which expresses the degree of precision of the ordering of the modeled events. This includes partial-ordered event accurate models, token-cycle accurate models, instruction-cycle accurate models, clock-cycle accurate models, clock-phase accurate models and so on.
2. *Data value detail*: which expresses the representation or format of data values specified in a model. Data values could be enumerated values, word-level values, bit-true representations, etc.
3. *Functional detail*: which expresses the level of detail in the functionality of a model, ranging from mathematical formulae to detailed intermediate operations (gate-level or instruction level).
4. *Structural detail*: which expresses the level of detail in the structure of a model, ranging from single-block code to multiple levels.

To ensure systematic composability of models, it is important to address how the composition is resolved along each of these dimensions. This is often achieved by creating *wrappers* around the existing library components to enable communication of data values between different modules and co-ordination between them. Since we are dealing with component descriptions in programming languages, the wrappers here refer to pieces of code that enable reuse of existing component models. Using programming languages, there are several ways in which such wrappers can be built. A common strategy is by using inheritance available in most object-oriented programming languages. In this approach, the wrapper is programmed by manually inserting code to align various design axes inside the inherited class. The component and the wrapper have a *common self* in this implementation; i.e. the wrapper and the component are the same object. As a result, the interoperability issues related to typing are resolved at compile time, and the wrapper and component have strong dependencies.

An alternative is to use a wrapper that, if needed, *delegates* to the design component. In this case, the component is not modified, and the wrapper and the component are two

distinct objects. Modules from different libraries can be imported as is, and dynamically placed in wrappers at runtime.

While inheritance can quickly achieve interoperability in some cases, it is not a recommended approach for many reasons. For one, it may actually hinder reusability in the long term. Figure 1(a) illustrates the UML class diagram of the typical problem

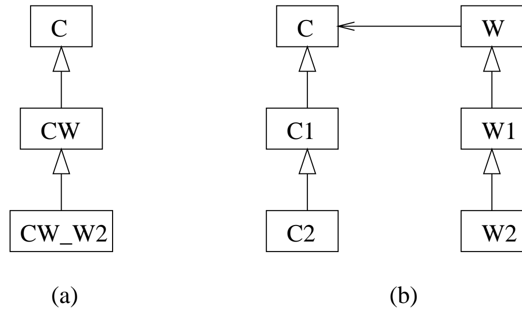

Fig. 1. Wrapper implementation strategies: (a) by inheritance (b) by composition

of inheritance-based composition (the *common-self* problem [21]). If a designer wants to reuse a component of class `C`, the class can be specialized by inheritance to a subclass `CW` to implement the wrapper functionality. Let us suppose that the behavior of the original class `C` is modified in the class `CW` by adding more functionality or by reimplementing a virtual function. If the class `CW` is to be reused in a different context, then it can be also inherited into a class `CW_W2` that implements more wrapper code to interoperate in the new integration context. The problem in this scenario is that all the three classes have a common self, and the original component has to be modified in every reuse context, via inheritance. Also, substituting an object of class `C` with a `CW` (which is legal in C++) may introduce subtle side effects. For instance, the wrapper code may modify the state of the component in ways that may not be obvious to the designer. Further, when reusing an inherited component, the class hierarchy must also be copied. By contrast, if in a reuse environment, the original component being reused is unaltered, then it keeps its identity distinct from the identity of the wrapper that contains the code for the interoperability. Figure 1(b) shows the UML diagram of how a wrapper hierarchy can be built for composition (the open arrow indicates an association). In this case, the wrappers are separate from the component object hierarchy, and the interoperability interface remains separated in the wrappers, and any call to functionality of the original component is delegated from the wrapper to the component. Researchers have suggested various ways of generating wrappers, or interface protocols. One such method which is often used in making various tools interoperate is by scripting. Examples of such scripting-based interoperability can be found in [19].

Among the prominent component composition frameworks Ptolemy [17] takes a very different approach to composition. Ptolemy views the interoperability between components as the problem of interoperating between their models of computations. As per our understanding, though not explicitly, but implicitly Ptolemy merges the four

dimensions into the notion of models of computation. Ptolemy defines "model of computation (MOC)" domains [12], and any component that can be described with Ptolemy belongs to one of the MOC domains. Ptolemy composes and simulates models by virtue of a hierarchy of *domain directors* that controls the simulation of a component encapsulated in its own domain [24, 13]. The global director can resolve the exchange of data values between the domains. Ptolemy's approach is clean and elegant, but it requires that the components be designed in the style of actors that conform to one of the Ptolemy domains. Also, it is not clear, how to use this framework in composing existing IP components, designed in multiple different languages, styles, and levels of abstractions, unless their MOCs are identified correctly.

2.2 Component Composition Frameworks: Desirable Features

A good Component Composition Framework (CCF) provides a composition language, and capabilities for dynamic composition, simulation and verification. The Composition language, either visual or textual, should be able to ask for components from the component library and should not have to worry about implementation types. The choice of types may be different between simulation and synthesis tasks. Automated type inference and type matching is useful not only along datavalue dimension but along other dimensions of interoperability. Hence, the framework must have automated support for selecting the correct type that makes the composition possible, with limited user intervention. However, just datatype matching is not enough, because interfaces may match in datatypes along the ports, but behaviors displayed at those ports may be drastically different due to differeing implementation, different models of computation etc.

The composition should be dynamic, in the sense, that one does not have to go through a recompile-test cycle when new components are added or replaced. Usually such a framework can be implemented in scripting, but that might sacrifice the efficiency of simulation, testing, coverage etc. So one must be able to simulate the composition of composed objects in compiled domain, without having to recompile the whole design for every change in the composition.

Given the importance of formal verification, a framework should at least be able to partially verify or enable constructively correct interface composition, or allow synthesis of the intervening protocol between the two interfaces.

In order to be able to compose components at different levels of abstraction, and/or models of computation, such meta information should be available about the components at a meta-level such that either the framework may use such meta-data for automated matching of components, or at least, can allow users to understand implications of composing two arbitrary components. We now examine the requirements imposed by these features.

Composition Languages: Unlike programming languages used for behavioral specification of components, the role of a composition language is to instantiate and connect the components. The component model describes the connections by dictating how and when things can be composed. A connection could also be thought of as a "relation" among components. Allen and Garlan [1] have proposed to consider connector separately from components in order to capture and isolate component interactions. There

have been many papers about methodologies and benefits for interface-based design [18], the separation of communication from computation. In component frameworks, this has to be pushed further as component interaction, typing and modeling dimensions have to be separated from computation [7].

Partial Typing and Typing Abstraction at the Architectural Level: This is the ability to be typeless at the composition level, where connections and relations should be loosely defined. This helps the conceptual design of SOCs by saving the effort of manually specifying every detail during the architectural exploration. For instance, when changing a bit width of a control word it should be abstracted in a "virtual connection" with a "virtual type". Connections can be abstracted by loose typing. This can even be pushed further as components can be abstracted, where multiple versions of the same thing could be hidden behind a general facade, with varying types of assumptions on the environments. In order to convert the virtual architecture into something that can be simulated, or into an implementation, depending on the way the framework is used. This includes data type matching and behavioral type matching, and can include the automation of the verification of the validity of a composition and interface verifications. Many co-simulation environments have been implemented, bus-functional models are often used for interfacing. Guerin et. al. [11] provide a good perspective for co-simulation in a SystemC backplane environment. They use a mixed-level interface as a transducer between protocols for communication on different levels of abstraction. SystemC and SpecC do programming level integration, while tools like System Studio [20] use graphical integration by linking ports. In software engineering research, architecture description languages (ADL) [14] have addressed parts of the problems of orthogonolizing component definition from component assembly and to solve typing mismatches [10]. Colif [6] provides an architecture description language (ADL), for describing topologies.

2.3 Platform-Based Design and Component-Based Design

Platform-based design (PBD) [25] is often defined as the creation of a stable core-based architecture that can be rapidly extended, customized for a range of applications, and delivered to the customer for rapid deployment. Platform-based design requires a "standard" architecture, to which components are interfaced, and to which wrapper can be generated and PBD often uses "put()/get()" interfaces. Generally, a PBD provides structure to pure component-based design by providing architectural constraints on SOC implementations.

Coral [3] and Colif [5] use standard architectures and effectively implement platform-based design in the architectural domain using the interface unit as the CoreConnect bus (or some other industry standard bus like an AMBA bus). These approaches use channels as connector, and use channel refinement to attach properties to the connector in order to pick an interface implementation from a library, that is compatible with the standard bus. On a more general note, bus interfacing, or wrapper generation are more complicated in general component frameworks than in platform frameworks due to the absence of interface standards and connectors.

3 Component Composition in BALBOA

The BALBOA [2] [9] component composition environment is a layered environment that provides a component model with introspection and *partial* typing capabilities. Components are composed dynamically using wrappers. These automatically generated wrappers use "split-level" interfaces to implement the composition rules, dynamic type determination and type inference algorithms. Split-level programming refers to system model generation and component programming in two different levels that are strongly connected by a matching class hierarchy and methods [16] . Split-level programming relieves the system engineers of programming artifacts and software engineering concerns and lets them focus on system architecture. The BALBOA component composition framework is used to build system models with an architectural perspective. The BALBOA framework is used for the following two different tasks:

1. **Architectural Design**: the system architect builds the overall system architecture by instantiating, connecting, configuring components, and establishing relationships among components;
2. **Component Design**: the library designer implements components or virtial components to populate the IP library using a programming language, such as C++. The implementation is restricted as much as possible on modeling a behavior or a structure.

The design of a library component has to be done by a designer who understands the language (e.g., C++/SystemC) The design of an architecture is done by a system architect, where the focus is on module instantiation and interconnection by using the architectural support in the component integration language (CIL). Figure 2 shows the layers in BALBOA. Languages are on the left side and the run-time structure on the right side. The description of the layers of Figure 2 is as follows.

The Architecture Definition Layer is where architectural structure is assembled from components using the Component Integration Language (CIL). The CIL is very close to an ADL but it implements a component model for component compositions and connections, an object model for object compositions, aggregations and associations. The CIL is built with Tcl, OTcl and TclCL extensions.

Tcl is used for the procedural and variable scripting basics. OTcl [23] provides object-oriented extensions to Tcl, to specify classes and instantiate objects. OTcl structures can be introspected by the "info" commands to query a class for its list of instances, an object for its type, list of attributes, and list of methods. TclCl is the link between OTcl and the C++ classes for combined manipulation. The CIL also uses a type system to abstract component types. Because the CIL is interpreted, we also refer to this layer as the *interpreted layer*.

The Component Definition Layer consists of a set of IP components stored in libraries. Any C++ class or object can be placed in this layer without it needing to derive from a specific C++ class. Ideally, this layer can accomodate C++ IP models in a range of libraries without affecting the implementation of the two other upper layers. This layer is also called the compiled layer.

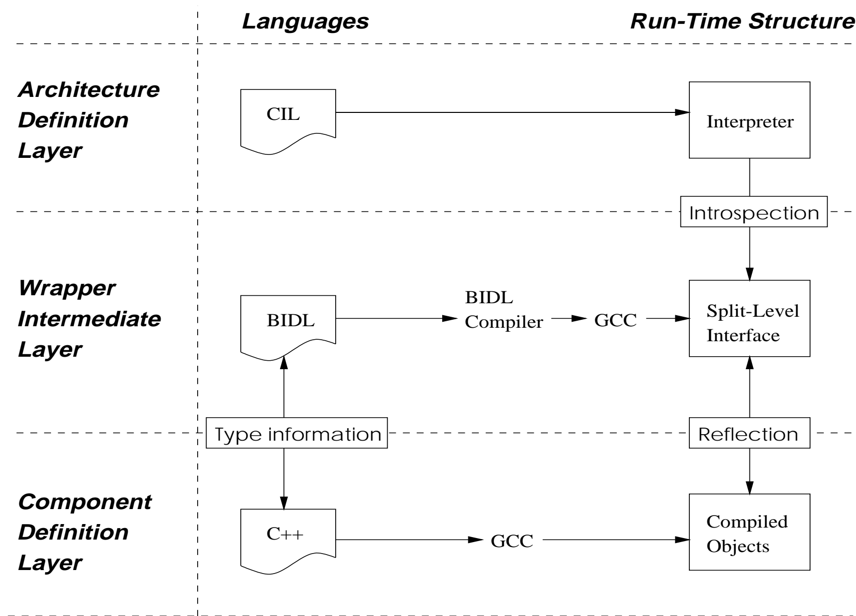

Fig. 2. Layering in the BALBOA environment: the languages are on the left side and the run-time structures on the right side of the figure

The Intermediate Wrapper Layer is the link between the interpreted and the compiled layer. Each C++ objects instantiated in the environment is contained and manipulated by a *split-level interfaces* (SLI). This wrapper provides the mechanism for manipulation of the compiled object by the scripting layer. The split level interface implements the *reflection* and the *introspection* capabilities [4] of the environment. The reflection is the capability of the split-level interface to read or write the attributes, and to invoke the methods of the compiled object. Introspection is the capability of the CIL language to query the reflected information of a component, and to understand its own structure.

The information that is being reflected and introspected is generated by the BALBOA Interface Definition Language (BIDL) compiler. The BIDL compiler translates and expands the description of the type of the component to a format that the interpreter can understand. The BIDL has a role similar to the CPP preprocessor- however it does not do macro expansions but a customization of the split-level interface framework specific to every component.

One of the novelties of the BALBOA environment is the separation between component definition and architecture elaboration through split programming to take advantage of weaker typing dependencies for typing abstractions at the architectural level. Typing abstraction means that it is possible to reduce the type dependencies of the strongly typed compiled C++ layer, through careful type management at the wrapper

level. The split-level interfaces can implement the type inference to keep the CIL description focused on component instantiations, compositions and connections.

BALBOA Typing: Our motivation in providing the hardware designer to design in a ”loose” typing environment comes from the observation that strict typing in C++ often results in excessive programming effort on the part of the system designer. This effort is particularly notable in case of using predefined IP libraries, where a component port type may be restricted to a subset of possible interface types. Our goal is to be to able to provide an environment that makes it easy to instantiate components and connect ports without specifying the C++ types completely. This is what we call *partial/loose* typing capability at the interpretive layer of our design environment. However, actual instantiation and running of a simulation cannot work without instantiating the correct concrete C++ types. We address this by type inferencing. In [8] we show that this problem is in general NP-Complete and we provide an efficient heuristic for solving the problem.

3.1 Using BALBOA

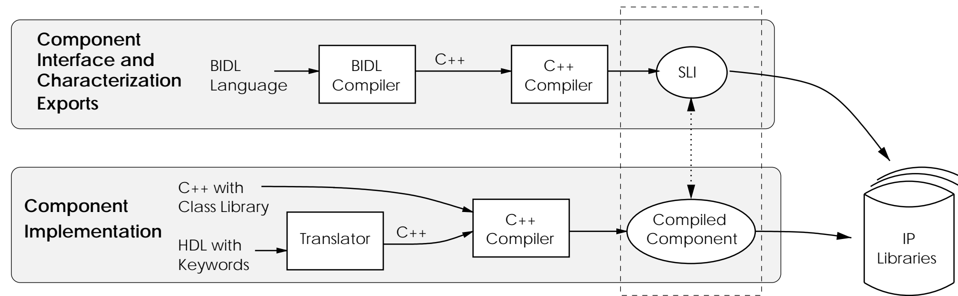

Fig. 3. Component design tool flow: write the component in C++, use the BIDL to characterize/export it and generate a SLI

Component Design: Figure 3 illustrates the tool flow and design process for the component designer. The lower part of the figure illustrates the flow for component implementations in C++. The upper part of the figure shows the flow for the component characterization and the exportation of the interface of the components to the interpreted domain. The BIDL compiler generates C++ code to create and configure the split-level interface of a component and to generate the type system information and the specific code for the delayed instantiation and delayed typing. The delayed type instantiation happens after the type inference problem is resolved as alluded to in the previoius subsection. The BIDL compiler also generates the object model configuration specific to the component. The principal steps for using BIDL to export a C++ class to the interpreter are the following: the designer uses the header of the class into the BIDL description and removes the part to be hidden from the interpreted domain. Keywords

are also added to configure the generation, such as component families, versioning and template classes handling and specification of available types. From the point of view of system architect, the component and the split-level interface can be the same entity, as shown in Figure 3 by the vertical dashed rectangle.

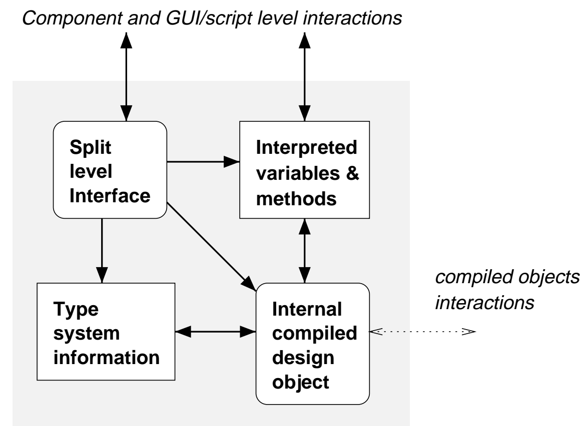

Fig. 4. Internal architecture for a BALBOA component

Figure 4 illustrates the internal architecture of a BALBOA component consisting of four blocks: the internal object (for example, such as a SystemC object or any other C++ object or component), the type system information (with an object model), the interpreted attributes/methods (that can be a reflection of the compiled attributes/methods), and the split-level interface routines. As shown in Figure 4, the split-level interfaces are the links between the interpreted domain and the compiled domain. Composition requests from the CIL script language are only interpreted in the SLIs. However, the simulation commands are delegated to the compiled components. Usually, the simulation control flow is kept only in the compiled layer because interpreted command execution in the SLI can be slow. However, the SLI layer can also interact with the simulation. For example, in our libraries we have a number of stimuli generators and monitors that use the interpreter control flow to compute stimuli and check assertion during the simulation.

Component Integration Language (CIL): The CIL is somewhere in between a module interconnection language and an architecture description language. This is because the CIL is used to build connections, and to build new components or compose attributes or behaviors to existing ones. The basic composition unit in CIL is an entity. For example, a component called `c1` is instantiated with the command:

```
Entity c1
```

This component can be composed of a subcomponent `c2` by the command:

```
Entity c1.c2
```

The result of this command is the instantiation of an entity named `c2` inside `c1`. The syntax for the composition is the dot "." operator, which is also used to navigate hierarchies. The CIL implements introspection [4], which is the capacity of an object to query itself to know its structure, attribute and methods. It is similar to self-inspection. For instance, Tcl provides introspection capabilities with the "info" procedure, and Java also provides introspection through the reflection packages. The BALBOA environment implements and extends these models to add introspection using a `query` method to the split-level interfaces. The following characteristics of a component can be queried: name, SLI type, C++ type, kind, attributes and methods. For example, the following query:

```
c1 query attributes
=> c2
```

returns the list of attributes for component `c1`. In this case, there is only the `c2` attribute that is returned as result of the command. This attribute is visible in the interpreted domain, but other attributes might be present in the compiled domain, but not visible if they were not exported. Complex commands can be built to query each subcomponent for information. For instance: The environment's use model for design assembly is built through introspection, looking for attributes or methods, and then introspecting them further to find out the composition possibilities according to an internal object model.

3.2 An Example of Component Composition in BALBOA

We illustrate use of BALBOA through the CIL level architectural composition of a compact packet switch example inspired by an example in the SystemC-2.0 distribution [20], shown in Figure 5.

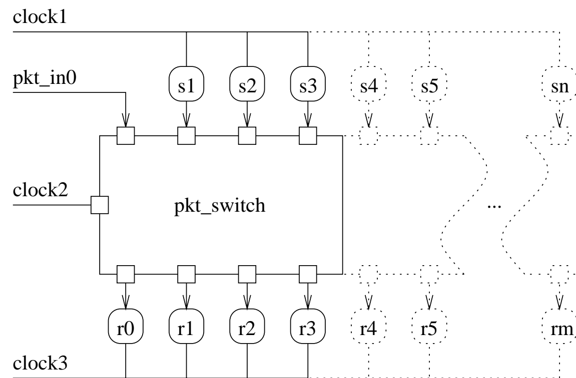

Fig. 5. Packet Switch CIL Example

Figure 5 shows a packet switching system with packet senders `s` and receivers `r`. The parameters of the switch can be configured, e.g., number of ports, up to *n* by *m* (4

by 4 on the figure). Secondly, the type of packet processed can be configured for the switch, senders and receivers. Modules for all possible types are stored in the IP library. Note that there is no sender for the first port of the switch since we assume another component to be connected to the first port.

```
1    set NUMBER_OF_PORTS  4X4
2    set PACKET_TYPE      Pkt
3    Pkt_Switch pkt_switch -number_of_ports $NUMBER_OF_PORTS
4    Signal     pkt_in0 -subtypes {$PACKET_TYPE}
5    Clock      clock1 -period 75 -duty_cycle 0.5 -start_time  0.0
6    Clock      clock2 -period 30 -duty_cycle 0.5 -start_time 10.0
7    Clock      clock3 -period 15 -duty_cycle 0.5 -start_time  0.0
8    connect pkt_switch.CLK to clock2
9    for {set i 0} {$i<$NUMBER_OF_PORTS} {incr i} {
10     if {$i>0} {
11       Sender   s$i -id $i
12       Signal   pkt_in$i
13       connect  s$i.CLK          to clock1
14     }
15     Receiver r$i -id $i
16     Signal   pkt_out$i
17     connect  s$i.pkt_out      to pkt_in$i
18     connect  r$i.pkt_in       to pkt_out$i
19     connect  r$i.CLK          to clock3
20     connect  pkt_switch.in$i  to pkt_in$i
21     connect  pkt_switch.out$i to pkt_out$i
22   }
```

Fig. 6. CIL listing for a 4 ports packet switch composition

Figure 6 shows the CIL listing for a switch topology composition. Line 1 sets a variable to ”4” for the number of ports. Line 3 instantiates the switch component, parametrized for 4 ports. Line 2 sets a variable to `Pkt` for the type of packets processed, and line 4 instantiates a signal with that sub-type. Lines 5-7 instantiate clocks for the senders, the switch and the receivers and lines 8, 13 and 19 connect the clocks to those components. The `for` loop on line 9 is parametrized to iterate for every port to instantiate a sender and receiver and connect them to the input and output ports of the switch. Lines 11 and 15 instantiate a sender and a receiver, lines 12 and 16 instantiate the signal connectors and lines 17, 18, 20 and 21 establish the connections between the components.

When the `pkt_in0` signal is connected to the switch, the split-level interface of the switch will pick the packet switch type with four ports that processes the `Pkt` packet type, among all possible switch implementation types in the library. The types for the signals, senders and receivers will be inferred to transmit and process the `Pkt` types. It is required that these types be defined in the libraries for the split-level interface to instantiate them. Of course, the same topology can be built using only C++. By comparison the description in C++ can be quite large (above 100 lines). One can find the

C++ listing of a nonparameterized version of this example in the SystemC downloadable distribution (in the examples directory) [20]. Because of the regular structure of the packet switch structure with respect to the number of ports and types, the usage of the CIL leverages the following advantages for flexibility and abstraction:

1. *Static parametrization for regular structures*:
   In this example, the design structure generation is parametrized with respect to the number of ports and use the static for loop of the CIL in the parametrization. The for loop instantiates and connects the surrounding signals, sender and receiver components for every port of the switch.
2. *Name expansion for regular structures*:
   Names are expanded by the interpreter with interpreted variable values. Component names for the signals, senders and receivers are expanded by the interpreter: `pkt_in$i` is expanded with the value of the iteration counter `pkt_in0`, `pkt_in1`, etc. for the design structure of the example.
3. *Type inference*: The components and connections are introspected by the environment and the split-level interfaces. In this example, the components will be picked by the enviroment to process the `Pkt` data type for the switch, the signals, and the senders and receivers.

This example illustrates how BALBOA composition is done in a typeless fashion and how it enables a separation of concerns of type compatibility from concerns of composition and the architectural structure. It also has the advantage of avoiding recompilation cycles when changing parameter values. BALBOA has been used in moderately complex designs that have been taken from high level specification all the way to harware level description [9].

4 Closing Remarks

Component Composition Frameworks (CCFs) represent an exciting development in the area of high-level modeling of complex SOC functionalities. A successful adoption of CCF is likely to have a direct impact on the succesful management of complexity of the new generations of SOC designs. However, there are several technical challenges that must be overcome. The chief among them are: ensuring inherent composability and reuse of SOC components. The problem extends beyond large scale program constructions in software engineering where several advances in architectural modeling and design environments have occurred. The challenge is due to the diversity of the computation models, levels of abstractions used and the notion of correctness applicable to SOC components. Thanks to advances in understanding of models of computation and their cosimulations (as exemplified by Ptolemy) an important aspect of the problem seems to have been addressed well. Challenges remain, however, in aspects related to encapsulation and reusability of components. The BALBOA framework addresses this aspect of the problem by essentially deconstructing the task of component creation from component composition. The underlying programming and automatic wrapper generation capabilities are built upon known advances in software engineering, namely,

reflection and introspection of the components and composition by delegation. The focus of our ongoing effort is to understand and develop techniques that can raise the level of abstraction used in interface composition, and exploit the system-level verification opportunities present in such an approach. We may view our approach as a bottom-up approach of SOC construction using reusable IP. On the other hand an enhancement in Ptolemy framework whereby Ptolemy actors can be mapped to existing IP components can be seen as a top-down approach. We are currently working in combining the two approaches to obtain a component composition framework that works both ways. We plan to enhance the top-down approach by raising abstraction a level further by means of aspect-oriented specification techniques[15].

References

[1] R. Allen and D. Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, July 1997.

[2] Balboa Project. Component Composition Environment. Home Page: http://www.ics.uci.edu/∼balboa.

[3] R. A. Bergamaschi, S. Bhattacharya, R. Wagner, C. Fellenz, M. Muhlada, F. White, W. R. Lee, and J.-M. Daveau. Automating the Design of SOCs Using Cores. *IEEE Design and Test of Computers*, 18(5):32–44, September-October 2001.

[4] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern Oriented Software Architecture: A System of Patterns*. John Wiley and Sons, 1996.

[5] W. Cesario, A. Baghdadi, L. Gauthier, D. Lyonnard, G. Nicolescu, Y. Paviot, S. Yoo, A. A. Jerraya, and M. Diaz-Nava. Component-based design approach for multicore socs. In *Proc. IEEE/ACM Design Automation Conf.*, 2002.

[6] W. Cesario, G. Nicolescu, L. Gauthier, D. Lyonnard, and A. Jerraya. Colif: a Multilevel Design Representation for Application-Specific Multiprocessor System-on-Chip Design. In *Proc. Int. Workshop on Rapid System Prototyping*, 2001. systemc.

[7] F. Doucet, R. Gupta, M. Otsuka, P. Schaumont, and S. Shukla. Interoperability as a Design Issue in C++ Based Modeling Environments. In *Proc. Int. Symposium on System Synthesis*, 2001.

[8] F. Doucet, S. Shukla, and R. Gupta. BALBOA: A Component-Based Design Environment for Composition and Simulation of System Level Models. *Submitted for Publication*, 2002.

[9] F. Doucet, S. Shukla, R. Gupta, and M. Otsuka. An Environment for Dynamic Component Composition for Efficient Co-Design. In *Proc. Design Automation and Test in Europe Conf.*, 2002.

[10] D. Garlan, R. Allen, and J. Ockerbloom. Architectural Mismatch: Why Reuse Is So Hard. *IEEE Software*, November 1995.

[11] P. Gerin, S. Yoo, G. Nicolescu, and A. A. Jerraya. Scalable and Flexible Cosimulation of SoC Designs with Heterogeneous Multi-Processor Target. In *Proc. Asia-South Pacific Design Automation Conf.*, 2001.

[12] E. A. Lee and A. Sangiovanni-Vincentelli. A Framework for Comparing Models of Computation. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, December 1998.

[13] E. A. Lee and Y. Xiong. System-Level Types for Component-Based Design. Technical Report ERL M00/8, UCB, Febuary 2000.

[14] N. Medvidovic and R. N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Trans. on Software Engineering*, January 2000.

[15] M. Mousavi, M. Chaudron, G. Russello, M. Reniers, T. Basten, A. Corsaro, S. Shukla, R. Gupta, and D. Schmidt". Using Aspect-GAMMA in Design and Verification of Embedded Systems. In *Proc. High Level Design Validation and Test Workshop*, 2002.

[16] J. K. Ousterhout. Scripting: Higher-Level Programming for the 21st Century. *IEEE Computer*, March 1998.

[17] The Ptolemy 2 Project, UC Berkeley, home page: http://ptolemy.eecs.berkeley.edu/.

[18] J. A. Rowson and A. Sangiovanni-Vincentelli. Interface-Based Design. In *Proc. IEEE/ACM Design Automation Conf.*, 1997.

[19] Simplified wrapper and interface generator (SWIG) home page: http://www.swig.org.

[20] SystemC. OSCI. Home page: http://www.systemc.org.

[21] C. Szyperski. *Component Software: Beyond Object Oriented Programming*. Addison-Wesley, 1998.

[22] Virtual Socket Interface Alliance, http://www.vsi.org .

[23] D. Wetherall and C. J. Lindblad. Extending Tcl for Dynamic Object-Oriented Programming. In *Tcl/Tk Workshop*, 1995.

[24] Y. Xiong and E. A. Lee. An Extensible Type System For Component Based Design. In *Sixth International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2000.

[25] CoWare N2C home page: http://www.coware.com.