

BALBOA: A Component-Based Design Environment for System Models

Frederic Doucet, *Student Member, IEEE*, Sandeep Shukla, *Senior Member, IEEE*, Masato Otsuka, and Rajesh Gupta, *Senior Member, IEEE*

Abstract—This paper presents the BALBOA component composition framework for system-level architectural design. It has three parts: a loosely-typed component integration language (CIL); a set of C++ intellectual property (IP) component libraries; and a set of split-level interfaces (SLIs) to link the two. A CIL component interface can be mapped to many different C++ component implementations. A type-inference system maps all weakly-typed CIL interfaces to strongly typed C++ component implementations to produce an executable architectural model. Thus, this amounts to selecting IP implementations according to a set of connection constraints. The SLIs are used to select, adapt, and validate the implementation types. The advantage of using the CIL is that the design description sizes are much smaller because the runtime infrastructure automatically selects the IP and communication implementations. The type inference facilitates changes by automatically propagating them through the design structure. We show that the inference problem is NP complete and we present a heuristic solution to the problem. We bring forth a number of issues related to the automation of reusable IP composition including type- compatibility checking, split-programming, and introspective composition environment, and demonstrate their utility through design examples.

Index Terms—Hardware/software co-design, system-on-chip, embedded systems, hardware description language (HDL), modeling, simulation, design reuse, interface design.

I. INTRODUCTION

RAISING the abstraction levels at which microelectronic system designs are entered and validated has a direct impact on the design quality and design time [1]. Consequently, recent research efforts in the area have been focused on the specification methodologies and languages for system-level

design. One prevailing view is to use C/C++, or a similar general purpose high-level programming language [2], to build custom architecture exploration and analysis frameworks. In the recent years, there have been several such proposals to help build digital hardware systems. Examples are SystemC [3], [4], SpecC [5], Ocapi [6], and others [7]–[10]. However, design composition is still tedious and reuse is *ad hoc* in the current compile-link-test methodologies. This is because C/C++ is a software implementation language, not hardware or system description language. A major barrier to its adoption for system-level design is that hardware designers need to understand significant software engineering issues related to “components” in software models. Often, such concerns are quite orthogonal to hardware system architectures and design issues, thus, actually adding to the time and effort in the system design process. For instance, strong typing requirements in C++ place a programming burden on the system architect to ensure that the component models and their interfaces are properly matched. The need to explicitly specify all typing information for components and their connections makes changes difficult. This is particularly notable when integrating predefined intellectual property (IP) components, where the availability of different data types for ports may be restricted, depending upon how the component has been modeled in C++. At the same time, there is a definite need to leverage the advantages of programming languages to quickly and accurately build executable system models.

In this context, our goal is to reduce as much as possible the software engineering and C++ programming problems faced by a system architect/integrator. To achieve this, a component composition framework provides reasoning capabilities and tools that enable a system designer to compose components into a specific application. These capabilities include selection and connection of the correct components, automated creation of correct interfaces, simulation of the composed design, testing and validation for behavioral correctness and equivalence checks. We built a system-level design environment called BALBOA which is a prototype component composition framework based on C++ class libraries approaches. BALBOA is used at the architectural level to design, evaluate and test functionality and performance. We use C++ for IP component definition, but we introduce a component integration language (CIL) for efficient architectural design. The CIL is a script-like language used to manipulate and assemble C++ objects incrementally and run simulations quickly without having to go through tedious recompilations cycles. CIL constructs provide an abstraction over C++ because they reduce the amount of typing information required to declare component

Manuscript received March 23, 2002; revised September 29, 2002. This work was supported in part by the Semiconductor Research Corporation through a Graduate Fellowship, in part by the Fonds de Recherche sur la Nature et la Technologies of the Province of Quebec pour la Nature through a Graduate Fellowship, in part by the Cal-MICRO program in partnership with Conexant, and in part by the National Science Foundation. This paper was recommended by Associate Editor R. Camposano.

F. Doucet was with the Center for Embedded Computer Systems, University of California, Irvine, CA 92697 USA. He is now with the Department of Computer Science and Engineering, University of California at San Diego, La Jolla, CA 92093–0114 (e-mail: fdoucet@ucsd.edu).

S. Shukla is with the Bradley Department of Electrical and Computer Engineering, Virginia Polytechnic Institute and State University, Blacksburg, VA 24060 USA (e-mail: shukla@vt.edu).

M. Otsuka is with the Design Verification Department, Technology Development Division, LSI Group, Fujitsu Ltd., Kawasaki, 211-8588, Japan (e-mail: otsuka.masato@jp.fujitsu.com).

R. Gupta is with the Department of Computer Science and Engineering, University of California at San Diego, La Jolla, CA 92093-0114 USA (e-mail: rgupta@ucsd.edu).

Digital Object Identifier 10.1109/TCAD.2003.819385

instances and connections. Using the CIL, a system architect can declare component instances with their interface types partially specified. We call this capability “partial-typing” (or loose-typing). Component integration can be done with partial types, but component execution needs full exact types. This is because a CIL design is simulated through an underlying C++ model (which cannot be done without instantiating the correct C++ objects).

In the BALBOA runtime environment, a type system automatically transforms the abstract CIL specification into a correct C++ implementation. The type system is responsible for 1) keeping track of the types of all specified components and 2) linking partially-typed component interfaces to fully-typed implementations. A type inference along the component connections in the design architecture is used to determine which C++ implementation types to assign to which CIL component interfaces.

Within the composition environment, a split-level interface (SLI) links a weakly typed interface to a strongly typed object implementation. The SLI provides the component-level implementation of the type system which includes the following information: 1) CIL interface type; 2) available C++ implementations; and 3) valid interface-to-implementation mappings with respect to all connections. The SLI provides a reflective layer, where all the interface- and implementation-type information is captured and accessible. This type information describes and implements the component composition rules. The system architect and the type inference can query this information to understand exactly what types are being manipulated. We call this the introspection capability and it is used by the type inference to produce a valid implementation selection.

The major contributions of this work are as follows. We have developed a component composition framework, which allows a clear separation between component definition and architecture elaboration. In the BALBOA environment, the level of abstraction is raised because the type dependencies between components are weaker when using the CIL than when using C++. The type inference lets system architects concentrate on architectural issues, rather than worrying about matching C++ types.

This paper is organized as follows. In Section II, we review definitions, background, and related work. Section III describes the BALBOA component composition environment, its usage, and implementation. Section IV presents the syntax and usage of the CIL and the BALBOA interface description language (BIDL). Section V describes the theory and implementation of the type abstraction and inference systems. In Section VI, we show how the CIL is used by a system architect as a front-end language. We present an implementation of a moderately complex design of an adaptive memory platform, called AMRM [11], in the environment and we discuss the results. We conclude in Section VII along with the description of the future work.

II. DEFINITIONS, BACKGROUND, AND RELATED WORK

The target of our work is microelectronic systems modeling for integrated design implementations. This integration (especially for single chip implementations) requires complete system-level simulations and verification of the systems. Accordingly, this work is based on advances in component

based design in software engineering, design specification languages and methodologies, and advances in type resolution in programming languages. We briefly review these advances.

A. Component-Based Design and Reuse

A component can be a piece of functionality implemented in software or as a dedicated piece of silicon hardware or a combination of the two. Components are units of composition and reuse—be it a function, object, library, or a complete program [12]. Usually, we can assume that a component will implement an interface to which another component will be connected. An interface is a “contract” between a component and its environment, a guarantee as a point of access. When establishing a connection, one component assumes that the interface of the other component implements the expected guarantees. Component technology is emphasized as a key element in the development of complex software systems [13]. Research in software engineering has demonstrated that the focus of the programming work is different when building components than when building architectures [14]. A component-based design (CBD) approach separates component definition from component composition. CBD is a bottom-up activity of assembling small components focused on one task into a more complex component with richer functionality. Reuse and parameterization have always been concerns when building components. Inheritance is used to share interface and behavior definitions, while polymorphism is used for defining several behaviors for the same interface. However, building architectures is done structurally, through instantiations and connections.

Reusability in architectures requires not only matching of interfaces, but also an ability to compose the functionalities in a way that correctly implements the end application. The difficulty of composition is due to the various ways in which the blocks can be represented, designed, and composed. This is especially true when considering microelectronic system modeling frameworks. The semantics of connections change between all levels of abstractions. At high levels, the amount of computation encapsulated behind an interface is much greater than at lower levels, but the structural design scope is much smaller. Different levels of abstraction can often be composed using protocol modules. The composability of models can be defined along a number of modeling dimensions, which describe what details are captured, to what level of accuracy, and how their modeling semantics are implemented (which syntax/control flow) [15].

B. System-Level Design and Architectures

System-level and hardware specification languages are active areas of research. Most approaches are based on programming language and raise the level of abstraction above register transfer level (RTL) into either the architectural and behavioral spaces design [16]–[19]. SpecC [5] and SystemC [4] are approaches that are based on the C/C++ programming languages. One of the problems is that many programming decisions and syntactical details that have to be addressed in C/C++ are independent of the system architecture model. Hardware designers and system integrators need not be concerned with inheritance, virtual function, genericity, and other tedious C++ specific constructs used

in module definition. Rather they should focus on characteristics specific to hardware such as bit width, propagation delays, regularity, etc.

There has been recent research about using component-based methodologies for system-level design. The results have shown significant reductions in design size and time required for automatic communication refinement. Approaches by Cesario *et al.* [20] and by researchers working on the Coral [21] framework start from a virtual architecture consisting of virtual component and virtual connections configured with sets of parameters. A microarchitecture implementation is generated by configuring the interface logic with respect to the configuration parameters. This can be done in two ways. The first one is comparison and matching of interface pin properties to find connection compatibilities and exact matches. Compatible pins can be connected with small interface logic like multiplexers, while pins with exact matches can be connected directly. The second technique is by channel refinement. A component will refine a read()/write() interface using method calls to communicate with a read()/write() interface implementing signal-level activity for bus transactions. The interface for the component will stay the same, albeit, it will take more cycle to perform a transaction. However, the interface will be connected to a bus instead of being connected directly to the recipient of the communication. This approach is sometimes called “transaction-level modeling” because transactions are decomposed from one event between components to many events on the bus. Both CBD strategies are platform-based design (PBD) approaches [22]. PBD is often defined as the creation of a stable core-based or bus-based architecture that can be rapidly extended and customized for a range of applications and quickly delivered to the customer for deployment. This requires a “standard” architecture or protocol to which components are interfaced. PBD provides structure to pure CBD through architectural constraints on system-on-chip (SOC) implementations. In other words, it provides the architectural template, and wrappers can be picked from libraries to implement the necessary protocols to which a component must conform.

Another well-known component framework is Ptolemy [23], which is targeted for system simulation and embedded software design. In Ptolemy, components are executed according to different models of computation. Components with their own thread of control are called “actors.” In an execution, the actors interact according to different models of computation (called “domains”). A model of computation is a collection of rules describing patterns for component executions and communications. Examples are data flow graphs, finite state machines, petri nets, etc. Domain-specific “directors” resolve the domain-dependent interactions and coordinate the actors from different domains. In Ptolemy, the different models of computation [24] are described using state machine [25]. The compatibility of two domains is determined by the output of the composition of their state machines. If they are compatible, the output will be a common state machine coordinating both domains.

All these approaches use components libraries, but they differ in the way the components are assembled. We identified three strategies for component integration. The first one is to compose components directly in the programming language used

for components definitions. This is the approach used with SystemC and Ocapi. In these cases, the code for setup and simulation is interleaved with the code for component definitions, thus making maintenance and reuse difficult. The second approach is to use a graphical capture tool with a notation of blocks and arrows (such as UML [26]) as a front end for code generation. The third approach is to use an architecture description language (ADL) [27], which orthogonalize component definition from system architecture composition [28]. ADLs are often domain-specific languages used to build an abstract model of a system and analyze it for schedulability, reliability, deadlock detections, etc. For system level-designs, ADLs have long focused on specialized tasks [29] such as processor descriptions [30].

Software implementations in C/C++/Java/Ada can be generated from several ADL models [31]. Code generation is useful because platform-independent architectural descriptions can be analyzed and targeted to specific machine. In this kind of ADL, component composition is usually done statically at the design time through code generation. Examples are Giotto [32] and Pecos [33]. There are other ADLs, such as Weaves [34], that do not use code generation, but dynamic composition, where components acquire references to each other at runtime. The advantage of dynamic composition over static composition is that the object relationships defined at runtime have weaker dependencies because they can be redirected, altered, and masked dynamically increasing flexibility and reuse [35].

There is another class of ADLs which are based on XML. These are declarative languages used for data exchange between tools. The MoML [36] XML dialect is used for describing and storing Ptolemy models. MoML incorporates system-level semantics to capture system architectures as actor topologies, hierarchies and relations. Such ADLs are used by programs because they are very easy to parse and generate. However, they are hard to read and usually not used by designers, as opposed to the other ADLs enumerated above.

Interoperability between design environments using ADLs can be difficult [15] because the ADLs often cover only some specific semantics of the “middleware”—the underlying computation model. In other words, an ADL that does not express all details of a design can be difficult to interoperate.

C. Split-Programming Techniques and Script Languages

Scripting is used to assemble components into applications where quick prototyping and flexibility are required. Scripting has been used for many years for component integration in computer-aided design frameworks as a sort of a module interconnection language. Script languages encapsulate APIs [37] behind an interpreted layer to reduce type dependencies. In the Tcl scripting language, variables have loose types because they can store any value that can be formatted back and forth to a string.

Split-level programming refers to architectural system integration and component programming in two different levels that are strongly connected by a matching class hierarchy and methods [37]. Split-level programming relieves the system engineers of programming artifacts and software engineering concerns specific to component implementation, and lets them

focus on system architecture. The key is to have the class hierarchies in multiple programming environments with “hooks” that enable their combined manipulation [38]. The network simulator (NS) system uses a split-programming model built on scripting to create a network simulation environment. In NS, there are two layers of programming facilities: one for building network components and the other for composing and simulating them. C++ is used for defining components that are used in an object Tcl (OTcl)-based scripting language to build and simulate a network topology model. The C++ classes that implement network components inherit from an OTcl base class that provides the hooks to be visible in the scripting layer [39].

Setting up a scripting environment to manipulate a C++ object can be cumbersome. A popular and efficient way to do it is by using a “wrapper generator” such as Swig. The Swig tool [40], [41] generates a wrapper around a C++ object to implements the script commands to instantiate/delete objects, access attributes, and invoke methods. The procedure is easy and efficient and the wrapper is transparent to the script user. However, in most CBD environments, the components are not simple objects, but they have complex internal architectures and behaviors [42]. Swig provides access to the C++ object, but it is not possible to configure and change the wrapper. Furthermore, Swig does not provide explicit support for a type system, runtime type construction, and type introspection. Also, there are a number of limitations when considering subtyping strategies, such as templates, parameterization, and inheritance. In the context of the BALBOA work, the partial-typing abstraction and type inference requires access to the wrapper and to a type system.

D. Type Systems

Most of the current system-level languages are strongly typed, with the exception of the Ptolemy framework which has an elaborate type system [43]. Ptolemy provides polymorphic actors, whose ports can have polymorphic types, i.e., they can be parameterized to take different combination of data types. Static type checking can determine the compatibility of a set of component interconnections. The polymorphism of actors is based on a lossless type hierarchy that forms a lattice. For instance, an integer type can be coerced losslessly into a double because double is higher in partial order than integer. This means that the value range of a double data type includes the value range of an integer data type. This value range inclusion property is used to put the data types in a lattice. Over this lattice, the static type resolution can be reduced to a solution of horn-clauses [44]. Thus, it is solved in linear time [43] by a fixed-point computation, which is a common way of computing type inference [45]. However, in order to be able to solve the type inference this way, Ptolemy requires the following. First, third-party actors need to conform to the polymorphic actor design principles—they must derive from a Ptolemy class or be wrapped in another Ptolemy actor. Second, port types are taken from the Ptolemy type library which conform to the type lattice structure. Unfortunately, for many silicon IP components (including legacy components), there are constraints on the availability of specific port types, either due to hardware design constraints or programming limitations. As a result, when

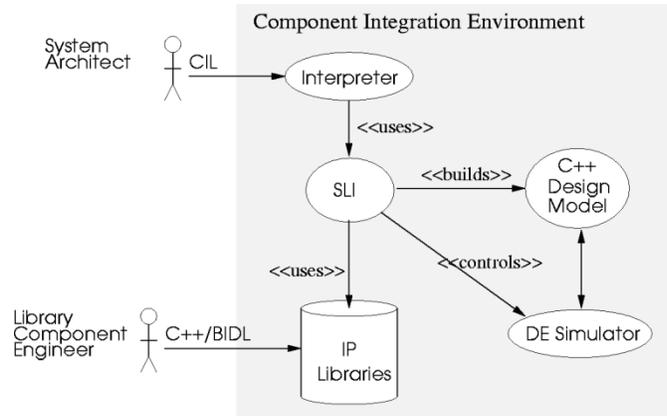


Fig. 1. BALBOA environment has two user roles: the system architect and the component library designer.

designing an environment that works with such preexisting libraries, one cannot assume the port types to fall in a lossless hierarchy.

In BALBOA, design components in the CIL are similar to Ptolemy actors. However, there are a few important differences. First, the discrete event computation model of Ptolemy is the closest to the model of computation in which BALBOA design models are simulated. SystemC and other C++-library-based modeling libraries implement the discrete event simulation semantics, and the components have this simulation kernel integrated into them. It is also possible to replace the prebuilt simulation kernel with implementation of other kernels that support other models of computation. However, our current focus is on enabling system designers with SystemC-like model integration. Second, our approach to typing is to enable composition of existing C++ based IP-libraries, which are implemented with arbitrary type systems of C++ and templated types.

III. BALBOA—COMPOSITION ENVIRONMENT

The BALBOA component-based design environment is used to build system-level models with an architectural perspective. The environment implements a split-programming model with an imperative CIL.

The BALBOA environment is used for the following two different tasks, as illustrated in Fig. 1.

- 1) **Architecture composition:** The system architect builds the system architecture by instantiating, connecting, configuring, and establishing relationships between components.
- 2) **Component definition:** The library designer defines and implements components to be used in the environment.

The design of an architecture is done by a system architect who can focus on module instantiation and interconnection by using the architectural constructs in the CIL. The design of library components has to be done by a designer who understands C++. The library designer also needs to export suitable component interface declarations (which may include behavioral interface, not discussed in this paper), using a specific interface description language.

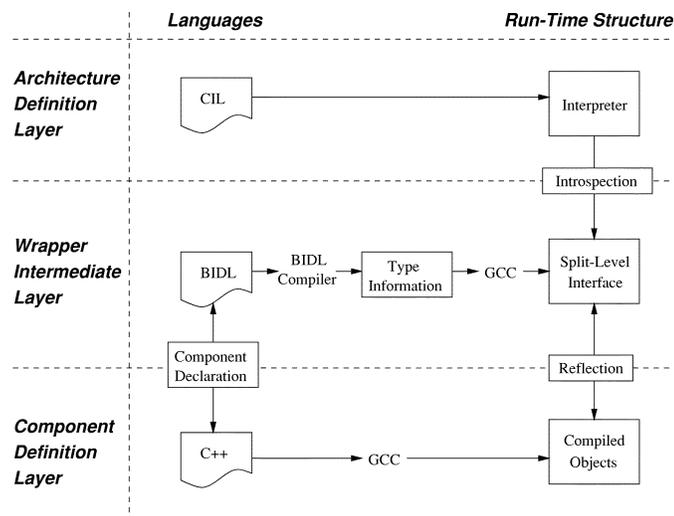


Fig. 2. Layering in the BALBOA environment: the SLI implements the typing abstraction and the type introspection capabilities.

A. Languages and Runtime Layering

In BALBOA, the languages and the runtime environment are layered. Fig. 2 shows the layers; their descriptions is as follows.

- 1) **The architecture definition layer** is where architectures are built using the CIL. It is an interpreted language based on OTcl [46] that implements a component model for instantiations, configurations, and connections. At this level, the type of a component is abstracted and a type management system is used to infer and instantiate the exact types required by the simulation model. We refer to this layer as the interpreted or scripting layer. The CIL is described in detail in Section IV.
- 2) **The component definition layer** is the bottom layer, where C++ components are stored in IP libraries. Classes in this layer need not be derived from a specific BALBOA C++ interface. Ideally, this layer can accommodate any C++ IP models in a range of libraries without affecting the implementation of the upper layers. This layer is also called the compiled layer.
- 3) **The intermediate wrapper layer** is the link between the interpreted and the compiled layer. Each CIL component is shadowed by a C++ compiled object that is contained and manipulated through an SLI wrapper. The SLI is the hook class that implements the reflection and the introspection capabilities [47] of the CIL.

Reflection is the capability of the SLI 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 being reflected and introspected is specified using the BALBOA interface definition language (BIDL) compiler. This includes the interface types, method signatures, as well as behavioral properties which may be usable by the environment for checking compatibility between components. The BIDL is described in Section IV.

The SLIs implement the type abstraction and inference to keep the CIL description focused on component instantiations, compositions and connections. Typing abstraction means that

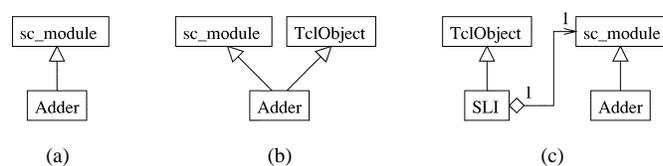


Fig. 3. How to implement necessary system and environment interfaces in C++: (a) hierarchy and concurrency system semantics in SystemC by inheritance; (b) add OTcl composition and manipulation by using multiple inheritance; and (c) separate system from environment semantics through dynamic composition.

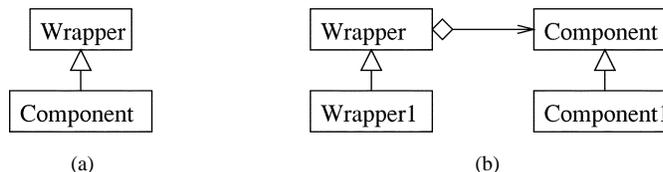


Fig. 4. Generic wrapping dependencies and strategies. (a) Strong compile-time inheritance dependency. (b) Weaker runtime composition dependency with introspection.

it is possible to reduce the type dependencies of the strongly typed compiled C++ layer, through careful type management at the wrapper level. The next subsection explains how the usage of an SLI reduces the type dependencies by using delegation.

B. Type Dependencies and Interfaces

The type abstraction in BALBOA is based on a programming tradeoff between inheritance and aggregation [35]. It is common to use inheritance to give design components the required interface to be manipulated in an environment. In BALBOA, we use aggregation (through the SLIs) because it weakens the type dependencies between a component and its integration environment.

Let us use an example to explain this. Consider that, if we define a class named `Adder` using the SystemC library, we have to inherit a class named `sc_module` that implements the interface to the simulation kernel. This is illustrated in Fig. 3(a) with an inheritance relationship. The inheritance dependency is specified in the class declaration and is resolved and checked at compile time. When instantiating an `Adder` to the environment, the object is also a `sc_module` because it has both interfaces. This property of inheritance relationships is generically illustrated in Fig. 4(a), where the instance of the wrapper and the instance of the component are the same. The flipside is that changing the interface of the wrapper also changes the interface of the component. Now, consider that we want to use this SystemC `Adder` in an OTcl environment. For this, it needs to implement the `TclObject` interface; therefore, we introduce the second inheritance dependency in Fig. 3(b).

The aggregation alternative is showed in Fig. 4(b). In this case, the instances of the wrapper and of the component are two different objects with distinct identities. In the BALBOA environment, we adopted this strategy. An example is illustrated in Fig. 3(c), where an `Adder` class inherits from the SystemC base class for the concurrency and hierarchy semantics, and an SLI inherits from the `TclObject` class for architecture composition and component assembly semantics. The SLI is the wrapper that aggregates the component and understands its semantics (that it manipulates a SystemC component).

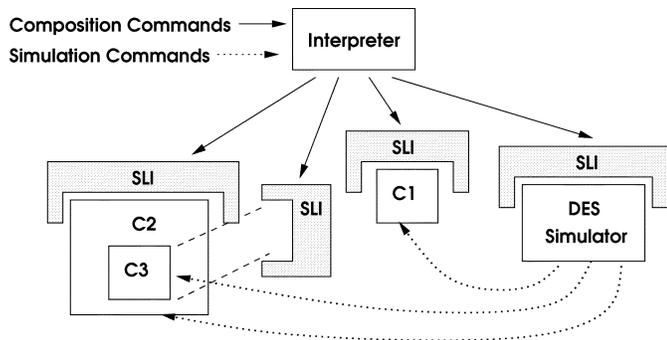


Fig. 5. Runtime environment structure. Each component has a SLI wrapper to interpret CIL script commands.

Unlike other implementations where a component inherits a class interface implementing the wrapper behavior, in our environment, the SLI aggregates the design component. It is deliberately designed to be different from the interface definition and implementation notions in Java. It often happens that two components providing the same functionality are implemented with different interfaces from different class libraries. However, we may want to use both components in the environment, for instance, if they provide different abstractions. Reuse by class interface precedes the design of the class, while reuse with the composition succeeds the design of the class [35]. In other words, instead of adapting the IP, it is better to adapt the wrapper.

Note that the SLI wrapper must not be confused with a bus or a protocol wrapper. These are used for translating or adapting a protocol into another one. In our case, the focus is on the issue of separating the semantics of the environment from the semantics of the design, and to use the SLI for typing and connectivity abstraction. Both are related because connectivity abstraction is indeed a part of the process of communication refinement, which can use bus wrappers.

In this case, the goal is to investigate type abstraction and interoperability by using object composition instead of class inheritance. The delegation provides a mechanism where compositional typing issues can be resolved dynamically and automatically by the environment (instead of being resolved at compile-time).

C. More on the Runtime Structure

Fig. 5 shows the relations between the interpreter, the SLIs, and the components at runtime. There are four compiled C++ design components: C1, C2, C3, and a compiled discrete event simulation kernel component, e.g., SystemC kernel. Every component has an SLI. Component C2 is composed of component C3, but the SLIs are not composed. The arrows represent the environment control flow. The full lines are the interpreted control flow, while the dashed lines are the compiled control flow. Usually, the interpreted control flow will be a set of composition commands that will be forwarded to the SLIs. The compiled control flow is usually the execution of the simulation. Note that the simulation semantics are part of the compiled control flow, since the simulation kernel interacts directly with compiled components. The simulation semantics are described in BIDL and loaded into the CIL type system as a set of component

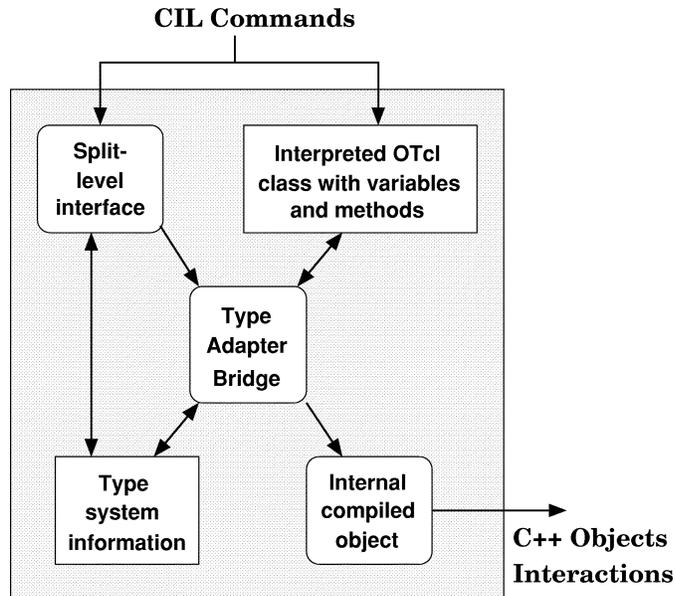


Fig. 6. Internal architecture of a BALBOA component.

types. The dashed lines in Fig. 5 are an illustration of the control flow from the discrete event simulator to the components. Nothing prevents mixing slow interpreted commands with the fast compiled simulation. When simulation speed is not an issue, using the CIL can be very expressive for validation and debugging. The CIL provides a number of stimuli generators, monitors, and assertion constructs that can be used for validation.

D. Internal Architecture of a Component

In the BALBOA framework, a component refers to all the layers on the right side of Fig. 2. A CIL component is an OTcl class with methods and attributes that are shadowed by all the compiled objects inside the shaded box in Fig. 6. The SLI manages the type information and implements the introspective and reflective capabilities. The type system reifies meta information, that being the C++ type of the internal design object (SystemC) and its nonfunctional properties. The type adapter bridge (TAB) is the only object that provides direct access to the internal compiled object. The SLI manipulates the internal object through the TAB interface. Both the SLI and the TAB will access the type system information. The SLI can work with partial type information, while the TAB can only work with the exact full type.

As described earlier, a CIL interface for a component can be mapped to many different C++ implementations. For each one of these possible mappings, a TAB specific to each C++ type will be created by the BIDL compiler. The type inference will select the appropriate TAB from a table in the SLI before the allocation of the internal design object.

IV. LANGUAGES IN BALBOA

In this section, we present the languages used specifically in BALBOA. IP components are implemented with C++ and, in theory, all C++ classes can be used in the framework. We will discuss the two other languages, CIL to assemble system architecture and BIDL to describe component interfaces.

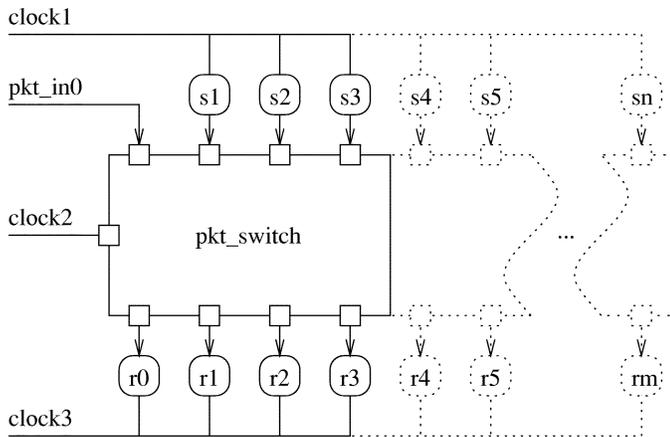


Fig. 7. Packet Switch CIL Example.

A. CIL

The CIL is used to build system architectures by instantiating and composing components. The CIL uses a type system to abstract component types. First, we illustrate how the CIL is used with a small example, and then show the abstract syntax for the language.

1) *Simple Packet Switch With the CIL*: System models built using programming languages can be parameterized in various ways [48]. However, it is often *ad hoc* and different from design to design. The CIL leverages the parameterization capabilities of programming languages by clearly defining three specific ways to do it. To illustrate this, let us consider the example of a very simple packet switch system shown in Fig. 7. It consists of packet senders s , receivers r , and a switch pkt_switch . The configurable parameters of the switch are the number of ports and the type of packets processed. A 4×4 configuration is shown in the figure. There is no sender connected to the first port of the switch because we assume another component will be connected there.

Fig. 8 shows the CIL listing for a switch-topology composition. We first start by setting a variable for the number of ports to 4×4 , and then instantiate the switch component with that parameter. The third line sets a variable to Pkt for the type of packets processed and then instantiates a signal with that subtype. The next three lines instantiate clocks for the senders, the switch, and the receivers. The “for” loop is parameterized to instantiate packet senders and receivers, and connect them to the input and output ports of the switch. No signals in the description have their data type specified. When connected, the signals’ SLI will pick the appropriate type (Pkt) among all possible implementation types in the library. The same process is done for the packet switch, where, in this case, it will pick the implementation with four ports that processes the Pkt packet format. The types for the senders and receivers will also be inferred to transmit and process the Pkt types. It is required that these types be available in the libraries for the SLI to instantiate them. Because of the regularity of the packet switch structure with respect to the number of ports and their types, the usage of the CIL provides the following advantages for flexibility and abstraction.

```

set NUMBER_OF_PORTS 4X4
Pkt_Switch pkt_switch -number_of_ports \
    $NUMBER_OF_PORTS
set PACKET_TYPE Pkt
Signal pkt_in0 -subtypes {$PACKET_TYPE}
Clock clock1 -period 75 -duty_cycle 0.5
Clock clock2 -period 30 -duty_cycle 0.5
Clock clock3 -period 15 -duty_cycle 0.5
connect pkt_switch.CLK to clock2
for (set i 0) (${i}<$NUMBER_OF_PORTS) (incr i)
  if (${i}>0) {
    Sender s$i -id $i
    Signal pkt_in$i
    connect s$i.CLK to clock1
  }
Receiver r$i -id $i
Signal pkt_out$i
connect s$i.pkt_out to pkt_in$i
connect r$i.pkt_in to pkt_out$i
connect r$i.CLK to clock3
connect pkt_switch.in$i to pkt_in$i
connect pkt_switch.out$i to pkt_out$i
}

```

Fig. 8. CIL listing for a four-ports packet switch composition.

- 1) *Control-flow parameterization for regular structures*: The CIL imperative control structures are used to build the system architecture. In this example, the design structure is parametrized with respect to the number of ports. The `for` loop instantiates and connects sender and receiver components and surrounding signals for every port of the switch.
- 2) *Name expansion for regular structures*: Component names are expanded by the interpreter with interpreted variable values. Names for signals, senders, and receivers are expanded such like in `pkt_in$i` variable i is replaced with the value of the iteration counter to `pkt_in0`, `pkt_in1`, etc.
- 3) *Type inference*: Types are assigned to partially specified components. The components and connections are introspected by the environment and the SLI selects a full type. In this example, the components will be picked by the environment to process the Pkt data type for the switch, signals, senders, and receivers.

The compactness of the CIL listing is due to the parameterization capabilities that enable a separation of concerns of type compatibility from concerns of composition of the architectural structure. Using control statements and name completion can be useful to complete component type names as well as instance names. Unlike code-generation approaches, it also has the advantage of avoiding recompilation cycles when changing parameter values. The same topology can also be built using only C++ with all subtypes and parameters explicitly specified. This yields longer descriptions that are less flexible as parameters change: 1) may need to be done throughout the design and 2) necessitate recompilation. An XML-based approach would also have yielded a lengthy description.

The CIL is like an imperative ADL, where checks are done on the composition as the design is constructed. A difference from most ADLs is that the CIL describes architectures without explicitly enumerating every component, connection, or type in the topology. It is actually very close to a domain-specific lan-

```

CIL → Tcl stmts|OTcl stmts|cil_commands

cil_commands → design_database_commands
               | type_database_commands
               | component_instantiation_commands
               | component_query_cmd
               | component_subtyping_commands
               | port_binding_commands
               | pointer_linking_commands

design_database_commands → design_db
                        ( query (delayed_cmds|toplevels|env_cmds)
                          | delayed_cmds (component_instance_name)?
                          | run_delayed_cmds_for_component (component_instance_name)? )

type_database_commands → type_db
                       ( list_all_possible_types
                       | type_inference_policy (CONTINUOUS|ONCE)?
                       | internal_obj_allocation_policy (LAZY|OPPORTUNISTIC)? )

component_type_name → Component | Signal | Port | Signal | <STRING>

component_instance_name → (component_instance_name.)*<STRING>

component_instantiation_command → component_type_name component_instance_name (parameters)*

component_query_commands →
    component_instance_name query (type|compiled_type|kind|attributes|methods|env_cmds)

component_subtyping_commands →
    component_instance_name get_subtypes
    | component_instance_name set_subtypes <TYPE>+
    | component_instance_name get_subtype <INTEGER>
    | component_instance_name set_subtype <INTEGER> <TYPE>

port_binding_commands → bind port_instance to signal_instance

pointer_linking_commands → link component_instance_name.pointer_name to component_instance_name

```

Fig. 9. Pseudo-BNF grammar of the CIL.

guage for system-level design. This shortens description sizes and helps the designer manage the complexity of describing architectures.

2) *Abstract Syntax in a Pseudo BNF Form:* Fig. 9 presents the abstract syntax of the CIL language. There are seven kinds of commands in the CIL. Let us enumerate and describe them. The first one is for the design database that keeps track of the components in the runtime environment. The second one is for the type database that is the heart of the type system. The type database can be queried to list all available types in the system, and to get/set the type inference options and the internal object allocation policies.

The third kind of command is to instantiate components. Each component instance has a type name that is not necessarily the same as the C++ type of the internal object. They may be very generic types such as `Signal`, `Port`, `Entity` etc. Components can be composed using the dot “.” operator.

The fourth kind of command is used for the introspection capabilities that are implemented through a `query` method in the SLI of every component. The environment uses the introspection capabilities to find the component characteristics, attributes, or methods. Introspecting them further finds out the architecture and the composition possibilities according to parameterizable internal object models. The following information can be queried:

- The interpreted type of the component- the OTcl type.

- The exact compiled type of the internal object. This can be different from the interpreted type because an SLI can aggregate many different C++ types for the same component. This corresponds to the type of the TAB.
- The kind of component. This is a characterization of the SLI orthogonal to its type. This is used as an aspect to group components that do not share the same component interface or implementation type. For instance, a `Signal` kind is used to group different C++ implementations of signal type under the same “kind.”
- The list of all accessible design attributes for the component in the interpreted domain. These include aggregation, associations (pointers), and both interpreted and reflected variables. Other attributes might be present in the compiled domain; however, they are not visible if they are not reflected.
- The list of all design methods and commands that are visible in the interpreted domain for the component type. These include both interpreted and reflected methods.
- The list of all environment commands that can be called for that component. These interpreted or reflected methods have no design modeling semantics, but only a meaning for the environment and supporting tools.

Note that 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.

```

BIDL → package_description

package_description → package name { (component_declarations)* }

component_declarations → (component|class) name {
    (kind_declaration)?
    (subtyping_declaration)?
    (subtyping_definition)*
    (attribute_declaration)*
    (method_declaration)*
    (class_for_internal_obj_declaration)* }

kind_declaration → kind name

subtyping_declaration → declare_subtypes { (subtype_type_name)+ }

subtyping_definition → define_subtypes { (subtype_value)+ }

attribute_declaration → <TYPE> name

method_declaration → <TYPE> name ((<TYPE> name)* )

class_for_internal_obj_declaration → use_class name for_subtypes { (subtype_value)+ }

subtype_type → component|class|<TYPE>

subtype_value → <TYPE> | <VALUE>

```

Fig. 10. Pseudo-BNF for an abstract syntax of the BIDL.

The fifth kind of commands is for subtyping. Type parameters is what enables partial typing in the CIL. For example, a `Port` type can be subtyped with a `bool` parameter. When all parameters are set, the CIL-to-C++ mapping can be done and validated. Note that “subtyping” here means parameterization. In CIL, it is possible to list and set all subtypes parameters for a component.

The sixth and seventh kind of commands in the CIL are to establish connections. Ports and signals can be bound and pointers can be linked to objects by the `bind` and `link` procedures, respectively. When using connection commands at the interpreted layer, type management is done by the SLIs. For example, when a pointer is set to a component through the `link_to` method, the runtime environment checks that the target component is of the right type for the association. For ports, the runtime environment will make sure that the transmitted data types are the same.

B. BIDL

The BIDL is used to describe the interfaces, type, subtypes, parameters, and characteristics of the internal object, and to “export it” to the interpreted domain. The BIDL compiler generates C++ code to create and configure the SLI and type adapters bridges. The BIDL was first inspired by the CORBA IDL [49] and has been extended and customized for the requirements of system-level modeling. The BIDL was developed incrementally as keywords were added to C++ class declarations. For example, the component designer can copy the header of a class into a BIDL description, remove the parts that should be hidden from the interpreted domain and add keywords for subtyping, kind and available instantiatable subtypes information. The BIDL compiler translates and expands the description of component types to a format that the interpreter can understand. The BIDL has a role similar to the CPP preprocessor. Instead of macro

expansions, it generates a custom type system extension, specific to every component type. These extensions are generated in C++, compiled, and placed into IP component libraries. As the type theory is developed, the BIDL syntax evolves as well, but this will be discussed in a future paper.

Fig. 10 shows the abstract grammar of the BIDL language. Component declarations can be done either in C++ or by using a neutral on the component definition language, such as the CORBA IDL. Kind, subtypes, attributes, methods, and class mappings are declared in the BIDL.

V. TYPE RESOLUTION IN BALBOA

A. Simple Parameterization Example: Fast Fourier Transform (FFT)

Fig. 11 shows the C++ code for an FFT-class interface, with the real and imaginary inputs and outputs ports, and a type parameter for the data widths. Fig. 12 shows a diagram of a sampling system using this FFT module. The connections between components are abstract, meaning that the user does not have to specify their types. In the BALBOA environment, a designer can query the libraries for available FFT implementation types and choose one. The type inference will propagate the chosen parameter to all components that are connected to the FFT. However, these other components need to have implementations for the chosen type parameter. If not, the type propagation will backtrack and request another different starting parameter. For instance, the downsampler may not have an implementation available in the library for a specific data width used as input to the FFT. In this example, backtracking is not difficult, but as the design grows in size, it is better that the system automatically handles this type propagation. In this section, we will explain how the CIL uses the type system to abstract C++ implementation types. We first consider small examples and then develop the type theory along the data type interface dimension.

```

template<class DATATYPE
class FFT {
    sc_in_clk      CLK;
    sc_in<bool>    data_valid;
    sc_in<bool>    data_ack;
    sc_out<bool>   data_req;
    sc_out<bool>   data_ready;
    sc_in<DATATYPE> in_real;
    sc_in<DATATYPE> in_imag;
    sc_out<DATATYPE> out_real;
    sc_out<DATATYPE> out_imag;
};

```

Fig. 11. Interface type declaration code for a FFT C++ component.

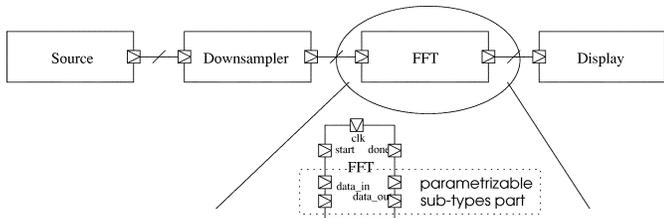


Fig. 12. Sample topology for a sampling and transformation system. The FFT is virtually connected to a sampler and a display. The virtual connection is implemented with the FFT C++ class of Fig. 11.

B. Data-Type Matching Problem

Data-type parameterization in C++ is often done using templates. In hardware design, this is especially convenient to express designs with various data widths. On compilation, a template type is turned into a concrete C++ type by parameter passing. This is useful in instantiating components whose interface and internal data types can be changed depending on the design requirements. For instance, we may have an adder with various implementations in the libraries. It can have implementations for integers, float, double, 4, 8, and 16 bit vectors and so on. When building architectural models, specific port types is a function of the architecture which may change as the design evolves. Ideally, one would like to instantiate the component without specifying all the type parameters, and resolve the detailed implementations automatically. For example, the system designer might just want to instantiate an adder, without specifying the input or output types.

The library component designer must specify what different C++ implementation types are available for this interface through a BIDL description. In the case of an adder, he/she must specify that it has been implemented for float, 8, 16, and 32 bit integers and so on. This information exported through BIDL is stored in the SLIs. The introspection facility can let the system architect know what type parameter values are available for a component instance. The architect may or may not choose values at instantiation. When the designer chooses to instantiate a component without specifying the types for each port, we say that the component is partially-typed.

1) *Split-Typing*: In the BALBOA environment, a partial type is also viewed as “split type” because it is managed through the SLI. A component has one type at the interpretive layer and can have multiple possible implementation types at the compiled level. In other words, a component has an interpreted type and

TABLE I
TYPE AVAILABILITY TABLE: ALL VALID COMBINATIONS OF TYPE PARAMETER VALUES FOR IMPLEMENTATIONS OF A CIL INTERFACE

In_1	In_2	Out_1	Out_2
float	float	float	bool
int8	int8	int8	bool
int16	int16	int16	bool
int32	int32	int32	bool
int64	int64	int64	bool
bool	bool	bool	bool

a compiled type. In the adder example, at the interpreter level, all the ports of the adder are of partial “Port” type. Based on the containment hierarchy and the connections to other components, type inference will associate specific C++ data types to each of the ports.

The following notation is used to illustrate the algorithms for incremental type inference. Let T be a set of all concrete C++ data types, let P be a set of ports, and let S be a set of signals in the design created in the CIL. In the interpreted layer, ports and signals are abstract types, but in the compiled layer, they must be mapped to a concrete type in T . Each port or signal is associated with a data type via mappings dt_p and dt_s , such that $dt_p: P \rightarrow T \cup \{\perp\}$, and $dt_s: S \rightarrow T \cup \{\perp\}$. A port is untyped if $p \in P: dt_p(p) = \perp$, and a signal is untyped if $s \in S: dt_s(s) = \perp$, where \perp denotes the fact that no concrete type has been specified or inferred yet. When the type inference is done, dt_p and dt_s must not map any element to \perp . If that cannot be achieved, the type-inference algorithm must detect and report this condition, and if possible back track to search for a new solution.

2) *Components*: In the CIL, a component c has a set of ports denoted by $ports(c) = p_{c_1}, p_{c_2}, \dots, p_{c_n} \in P$, where n is the number of ports. Component c is said to be “polymorphic,” if there are many compiled versions of c with ports of different types. The function dt_p has a limited choice in assigning $p_{c_1}, p_{c_2}, \dots, p_{c_n}$ to the available compiled types. One can view this as a choice of assignment to a vector of ports $\langle p_{c_1}, p_{c_2}, \dots, p_{c_n} \rangle$, from one of the possible rows from a type availability table of ordered rows $T_c \subset T \times T \times T \times \dots \times T$, each row corresponding to a compiled version of c . The rows of the table are filled through the BIDL description. Signals are used to link ports and are parameterized by the data type they carry. A signal must be assigned the same data type as the ports it is connected to. Hence, signals are constraints in the port assignment problem.

3) *Simple Matching Example*: Let us consider the simple adder example with 4 ports, in_1, in_2 , as input ports and out_1, out_2 as output ports. The availability of library implementation of the adder is illustrated in the type availability table of Table I. The table list all available combination of type parameter values, each row corresponding to a specific C++ implementation type.

Let us consider the following scenario. An adder is declared in the CIL layer without any port types, then $dt_p(p)$ is undefined for all ports p . As types are propagated through the design, it turns out that in_4 gets instantiated to bool. Now, the type management system will immediately recognize that all implementation in the table have this parameter value, so no con-

clusion is available. Let us assume next that in_1 gets typed to int_{16} . The SLI can now match the row: $(\text{in}_1, \text{in}_2, \text{out}_1, \text{out}_2) = (\text{int}_{16}, \text{int}_{16}, \text{int}_{16}, \text{bool})$. Then, it will be inferred that in_2 and out_1 should also be of type int_{16} . At this point, any signal that is connected to in_2 and out_1 will have the type propagated to them as int_{16} . This means that $\text{dt}_p(\text{in}_1) = \text{int}_{16}$, $\text{dt}_p(\text{in}_2) = \text{int}_{16}$, and $\text{dt}_p(\text{out}_1) = \text{int}_{16}$.

C. Formulation of the BALBOA Type Inference Problem

Given a design with a set P of ports, a set S of signals, and the partition of P into k disjoint sets, where k is the number of components and the partition of P is disjoint because, as components do not share ports, the type inference problem is as follows. For each component c , with its port vector $\langle p_{c_1}, p_{c_2}, \dots, p_{c_n} \rangle$, assign a row from its type availability table T_c , such that if there is a signal $s \in S$, which connects a port p_{c_i} in component c to another port p_{d_j} of a component d , then the type assigned to p_{c_i} and to p_{d_j} must be the same. This restriction makes the problem complex, and we show here that the problem is NP complete.

Theorem 5.1: The BALBOA type inference problem is NP Complete.

Proof: Given a type assignment for all the ports, it is easy to check in polynomial time that the assignments in the type tables are correct, and the ports connected via signals have the same type. Hence, the problem is clearly in NP.

For the NP-hardness proof, we reduce the problem of one-in-three monotone 3SAT [50], to the BALBOA type-inference problem. 3SAT is the following problem. Given a set U of Boolean variables and a collection of disjunctive clauses over U , such that each clause is a disjunction of exactly three literals, find if there is a truth assignment of the variables in U , such that all of the clauses are satisfied. One-in-three 3SAT, is a special case of 3SAT problem, where the truth assignment of U has the restriction that if x_i and x_j are two literals appearing in the same clause, then both cannot be assigned the truth value of one. Monotone one-in-three 3SAT, has the further restriction that no negated literal appears in any of the clauses.

Given an instance of monotone one-in-three 3SAT, for each clause c , which has three literals $x_{c_1}, x_{c_2}, x_{c_3}$, one can create a table T_c , with exactly three rows $\{(1, 0, 0), (0, 1, 0), (0, 0, 1)\}$. Given the set of vector of variables in c , $\langle x_{c_1}, x_{c_2}, x_{c_3} \rangle$, one has to assign one of the rows in table T_c with the restriction that if the variable that appears as x_{c_1} in c also appears in another clause d as x_{d_i} , then the choice of the rows from T_c and T_d must be such that the assigned truth values are the same for x_{c_1} and x_{d_i} . This is a version of the BALBOA type-inference problem and, hence, it is at least as hard as the monotone one-in-three 3SAT, which is known to be NP complete [50].

Given that the problem is NP complete, we use heuristics to solve the problem. Moreover, the problem is solved incrementally because components are added or deleted in the CIL—the type assignments keep changing. The heuristic implemented in the BALBOA type-inference uses a delaying mechanism which delays the type instantiation and allocation of components until all type parameters associated with a component are resolved. The resolution is done on-the-fly, as components are added, type parameters get propagated to partial types.

D. Heuristics for BALBOA Type Inference

There are multiple ways of solving these problems, including dynamic programming, local search [51], etc. A sequence of natural join operations may take a long time, since the different T_c s have different sizes. The order in which the joins are computed can be optimized using dynamic programming; hence, we will follow that approach. However, since we build models incrementally, when a new component is added and connected, the new required join may not be in the order that the dynamic programming would have yielded. In the current implementation of the BALBOA environment, the runtime type inference mechanism may be summarized as follows. We first verify compatibility of types when connections are set between two components. We join the type availability tables of both components, renaming the connected ports to the same name if necessary. If the join returns empty, the components are not compatible; if it is not empty, the system will remember the result. Since the SLI allocates the component, it can decide to delay the allocation until the type parameterization is resolved. Now, considering this join result as the type availability table for this pair of components, we then select a new component that is connected to one of these, and apply the procedure.

If the result of the final join is empty, the architecture described in the CIL cannot be instantiated with the available implementations. If it is singleton, then we have found a unique combination of type parameters that can be instantiated. If the final join yields multiple rows, we have to choose a type parameter combination from the final table.

Fig. 13 shows a simplified version of the type-inference heuristic. The *CheckTyping* heuristic chooses two components that are connected, both with small type availability table sizes. It does a join locally between the two components to assess that the types are compatible. While the designer is using the CIL, this join is used to propagate type parameters as the architecture is built. All connection information is passed to the join procedure as `Cnx`, which is used to decide which columns of the tables to run the join on.

In the current implementation, the runtime environment does not build the join tables explicitly, but walks the architectural graph and builds the join tables implicitly in the SLIs. In this case, the runtime environment does an incremental version of this procedure by remembering all the intermediate join results through the SLIs. In future works, we will study optimizations of this procedure, including caching joins, incremental joins, etc.

VI. IMPLEMENTATION AND RESULTS

This section shows the usage of the CIL for a moderately complex real design example. The AMRM is an adaptive cache memory system [11] that can have its configuration changed dynamically. For instance, associativity and line size can be configured by special processor instructions. The hardware part of the design is a regular cache subsystem with a modified controller for cache adaptation.

Fig. 14 shows the outline of the procedure we followed for component integration and communication refinement. At each step, we refined both the component and the connector, but we

```

/* Type Inference Heuristic: Propagates Type Parameters to Components in an Architecture */
CheckTyping(A, C, Cnx): At
  Input:      Untyped architecture A, of component C, and connections Cnx
  Output:     Typed architecture At
  Intermediate: Open:Set, Done:Set
1:   for all (ci, cj) ∈ C
2:     join_result = RunNaturalJoin (ci, cj, Cnx)
3:     if (join_result == empty)
4:       Backtrack(ci, cj)
5:     else
6:       Put (ci, cj) in Done;
7:       Put all components that are connected to ci and cj in Open;
8:       for all ck in Open {
9:         Add all components connected to ck that are not already in Done to Open;
10:        join_result = RunNaturalJoin(join_result, ck, Cnx);
11:        if (join_result == empty)
12:          Backtrack(join_result, ck)
13:        Move ck from Open to Done;
14:      } /* end for */
15:    } /* end else */
16:  } /* end for all */
17:  if join_result has a singleton row,
18:    Compiled model can be allocated;
19:  if join_result has multiple rows
20:    Designer needs to chose one row.

```

Fig. 13. Simplified type inference heuristics implemented in the BALBOA runtime environment.

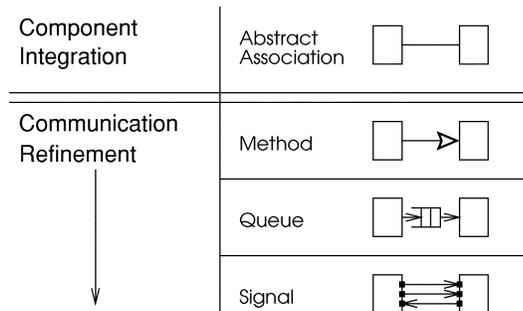


Fig. 14. AMRM models in different levels of abstraction for the components and connectors.

encapsulated the changes behind a CIL interface. Each box represents a memory hierarchy level, encapsulating the controller and the memory array, and the line between the two boxes represents a connection between the levels. At the top most level, we assume that there is a connection between the memory levels but there are no assumptions on its type or implementation. Communication refinement yields several versions of the component and the connector. Fig. 15 shows both the UML class diagrams and the block diagrams for the component integration and the communication refinements.

The script listed in Fig. 16 shows the CIL file used at all refinement levels: we start by instantiating two cache components named L1 and L2, and a memory component named Mem. The last lines invoke method calls to the interface of the components to establish the connections. These methods are reimplemented as the abstract connection is refined.

A. First Refinement

In the first refinement, the connector is a pointer referring to the lower level of memory hierarchy. Method invocations along

this pointer implement message passing in a sequential model. In the class diagram of Fig. 15(a), Memory_Base is the base class for the Cache and Memory classes. The Memory_Base has read and write virtual methods that are implemented in the Cache and Memory classes to implement the component behaviors. The Cache class has an association (pointer) named lower_memory that is used to navigate to the lower level of memory. For example, on an L1 cache read miss, the L1 cache will use this association to call read method of L2 cache. The block diagram in Fig. 15(d) shows how these lower_memory associations implement the control flow between the two levels of cache and the main memory. The procedure listed in Fig. 17 sets the association pointers between two caches. It is an OTcl method of the Cache_Ctrl component that sets its own pointer (by \$self) to the lower memory level.

B. Second Refinement

In the second refinement, we change the pointer for two queues: one for the requests and one for the answers. The class diagram on Fig. 15(b) illustrates this change. The refinement also introduces concurrency with the addition of a reactive process named proc to the Cache and Memory classes. These processes are triggered by events on clock input ports and transitively call the read() and write() methods. The script on Fig. 18 lists the procedure to connect two caches together with queues as link objects. The first lines instantiate the queues. The data types of the queues will be set according to the types of the association pointers to which they are connected. The other lines establish the associations between the caches and the queues. Fig. 15(e) shows the architectural view, where each cache level is separated by two queues.

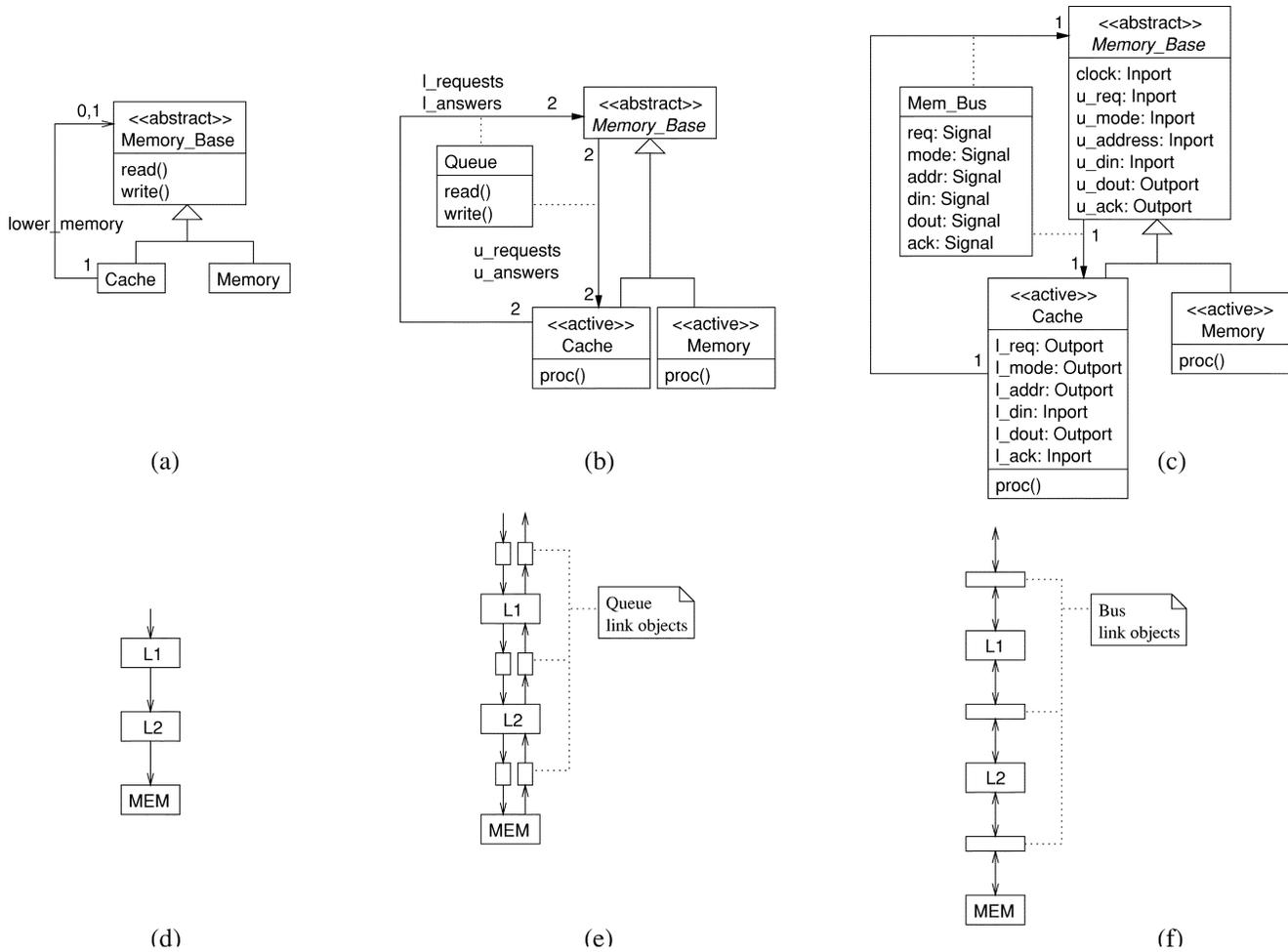


Fig. 15. AMRM component integration models with communication refinement: the upper row is for the class diagrams, and the lower row is for the corresponding block diagrams: (a) and (d) pointer connectors, (b) and (e) queue connectors, and (c) and (f) signal connectors.

```

# Load the AMRM component library
load ./libamrm.so

# Component instantiations
Cache L1
Cache L2
Memory Mem

# Procedure calls to connect components
L1 connect_cache2cache L2
L1 connect_cache2mem Mem

```

Fig. 16. Top-level architecture file for the AMRM structure.

```

Cache_Ctrl instproc connect_cache2cache { l_c } {
  link ${self}.lower_memory to $l_c
}

```

Fig. 17. Connection procedure.

```

Cache_Ctrl instproc connect_cache2cache {level2} {
  # Queue instantiations
  Queue l${level2}_req
  Queue l${level2}_ans

  # Connect queues to the upper cache
  link cache_l${self}.l_requests to l${level2}_req
  link cache_l${self}.l_answers to l${level2}_ans
}

```

Fig. 18. Refined connection procedure.

C. Third Refinement

The lowest level of abstraction in our AMRM models uses signal communications. Fig. 15(c) shows the class diagram for this model. The behaviors of the queues are still in the design, but is refined through ports beginning by “l” for the lower memory, and by “u” for the upper memory. These ports are bound to the Mem_Bus link class, which encapsulates all signal objects. Fig. 15(f) shows the block diagram with the memory hierarchy and the buses. The script in Fig. 19 lists the procedure to connect two caches through a bus. We now use the bind command for the port object instead of the link command for pointers. The third line instantiates a cache bus named cb. The remainder of the listing individually connect the ports of the upper and the lower cache to the bus signals.

D. Code Generation Ratios and Discussion

Table II shows the design statistics of the file sizes and code-generation ratios for the various AMRM implementations. As we refine the models, the script sizes grow larger, but not the number of C++ classes (except for the queue data type class in the second refinement). This is because the granularity of the communications gets smaller and there are more connections to be established.

```

Cache_Ctrl instproc connect_cache2cache {L_Cache}
{
  # instantiate a cache bus
  Cache_Bus cb
  # connect bus signals to upper cache
  bind ${self}.l_req to ${cb}.req
  bind ${self}.l_mode to ${cb}.mode
  bind ${self}.l_addr to ${cb}.addr
  bind ${self}.l_dout to ${cb}.din
  bind ${self}.l_ack to ${cb}.ack
  bind ${self}.l_din to ${cb}.dout

  # connect bus signals to lower cache
  bind ${L_Cache}.u_req to ${cb}.req
  bind ${L_Cache}.u_mode to ${cb}.mode
  bind ${L_Cache}.u_addr to ${cb}.addr
  bind ${L_Cache}.u_din to ${cb}.din
  bind ${L_Cache}.u_ack to ${cb}.ack
  bind ${L_Cache}.u_dout to ${cb}.dout
}

```

Fig. 19. Refinement with signal connectors.

TABLE II
DESIGN STATISTICS OF AMRM MODELS: CODE GENERATION
RATIO AT HIGHER ABSTRACTIONS

Level of Abstraction	CIL # lines	C++/CIL ratio	BIDL # lines	IP vs. Generated C++ lines
Methods	< 30	812/30 (27:1)	60	812/809 (1.01)
Queues	< 40	1512/40 (37:1)	84	1512/1002 (1.51)
Signals	< 150	1437/150 (10:1)	87	1437/880 (1.63)

The programming efforts to write BIDL files in the experimentations were nonintrusive and of low effort. We used the parts of the header of the classes we wanted visible to the interpreter and we added characterizations as behavioral or structural components. The ratio of the IP versus generated code sizes shown is increasing as the abstraction is lowered. However, the size of the BIDL file and the generated code does not grow linearly with the size of the IP code. The environment does not take design decisions for the designer in the communication refinement, except for the type propagation. Work is in progress to investigate what ratios of code generation are obtained in different design contexts, and to minimize the size of the SLIs.

The CIL provides a high-level view focused on the components and the connectors. Parameterization with control flow, name concatenation, and type inference parameterization with no recompilation are definite advantages over using only C++. The AMRM design example shows a refinement using the component modeling capabilities of the CIL extended with connection-method customizations. A future enhancement will be to isolate the connector from the connection method, so that the type inference can choose a connector according to the set of communication patterns.

VII. CLOSING REMARKS AND FUTURE WORK

Component composition frameworks represent an exciting development in the area of high-level modeling for system-level design. A successful adoption of those frameworks is likely to have a direct impact on the successful 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. Advances in the understanding of cosimulation and models of computation are important aspects of the problem that have been addressed well. Challenges remain in aspects related to encapsulation and reusability of components. The BALBOA framework addresses this aspect of the problem by deconstructing the task of component creation from component composition. Our approach is a bottom-up approach of SOC construction using reusable IP. The underlying programming and automatic wrapper generation capabilities are built upon software engineering techniques, namely, reflection and introspection of the components and composition by delegation. The focus of our ongoing effort is to understand and develop techniques to raise the level of abstraction used in interface composition and exploit to the system-level verification opportunities present in such an approach. For example, in the type inference arena, the data-type matching is just a starting point. Currently, a behavioral type system is being developed to check the functional validity of composition of virtual components.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for their constructive comments and suggestions. The authors would also like to thank J.-P. Talpin, R. Jerjurikar, C. Pereira, and S. Tauro for their technical feedback, help, and suggestions.

REFERENCES

- [1] Semiconductor Industry Association, International Technology Roadmap for Semiconductors (2001). [Online]. Available: <http://public.itrs.net/>.
- [2] R. K. Gupta and S. Y. Liao, "Using a programming language for digital system design," *IEEE Design and Test of Comput.*, pp. 72–80, Apr.–June 1997.
- [3] S. Liao, S. Tjiang, and R. Gupta, "An efficient implementation of reactivity in modeling hardware in the scenic synthesis and simulation environment," in *Proc. IEEE/ACM Design Automation Conf.*, 1997, pp. 70–75.
- [4] OSCL, SystemC [Online]. Available: <http://www.systemc.org>.
- [5] D. Gajski, J. Zhu, R. Domer, A. Gerstlauer, and S. Zhao, *SpecC: Specification Language and Methodology*. Norwell, MA: Kluwer, 2000.
- [6] P. Schaumont, S. Vernalde, L. Rijnders, M. Engels, and I. Bolsens, "A programming environment for the design of complex high-speed ASICs," in *Proc. IEEE/ACM Design Automation Conf.*, 1998, pp. 315–320.
- [7] L. Semeria and A. Ghosh, "Methodology for hardware/software co-verification in C/C++," in *Proc. High-Level Design Validation and Test Workshop*, 1999, pp. 67–72.
- [8] G. D. Michelli, "Hardware synthesis from C/C++ models," in *Proc. Design Automation and Test Eur. Conf.*, 1999, p. 80.
- [9] C. Weiler, U. Kebschull, and W. Rosenstiel, "C++ base classes for specification, simulation, and partitioning of a hardware/software system," in *Proc. CS Workshop on VLSI*, 1995, pp. 777–784.
- [10] R. Roth and D. Ramanathan, "A high-level hardware design methodology using C++," in *Proc. High-Level Design Validation and Test Workshop*, 1999, pp. 73–80.
- [11] AMRM, Adaptive Memory Platform (2000) [Online]. Available: <http://www.cecs.uci.edu/~amrm>.

- [12] M. D. McIlroy, "Mass produced software components," in *Software Eng.: Rep. Conf. Sponsored by NATO Sci. Committ.*, Oct. 1968, pp. 138–155.
- [13] J. Hopkins, "Component primer," *Commun. ACM*, pp. 27–30, Oct. 2000.
- [14] D. E. Perry and A. L. Wolf, "Foundations for the study of software architecture," *ACM Software Eng. Notes*, vol. 17, no. 4, pp. 40–52, 1992.
- [15] F. Doucet, R. Gupta, M. Otsuka, P. Schaumont, and S. Shukla, "Interoperability as a design issue in C++-based modeling environments," in *Proc. Int. Symp. Syst. Synthesis*, 2001, pp. 87–92.
- [16] D. Verkest, J. Cockx, F. Potargent, G. Jong, and H. D. Man, "On the use of C++ for system-on-chip design," in *Proc. CS Workshop on VLSI*, Apr. 1999, pp. 42–47.
- [17] S. Vernalde, P. Schaumont, and I. Bolsens, "An object oriented programming approach for hardware design," in *Proc. CS Workshop on VLSI*, Apr. 1999, pp. 68–75.
- [18] K. Wakabayashi and T. Okamoto, "C-based SoC design flow and EDA tools: An ASIC and system vendor perspective," *IEEE Trans. Computer-Aided Design*, vol. 19, pp. 1507–1522, Dec. 2000.
- [19] J. Zhu, "MetaRTL: Raising the abstraction level of RTL design," in *Proc. Design Automation and Test Eur. Conf.*, 2001, pp. 71–76.
- [20] W. Cesario, A. Baghdadi, L. Gauthier, D. Lyonard, G. Nicolescu, Y. Paviot, S. Yoo, A. A. Jerraya, and M. Diaz-Nava, "Component-based design approach for multicore SoCs," *Proc. IEEE/ACM Design Automation Conf.*, pp. 789–794, 2002.
- [21] 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 Test*, vol. 18, pp. 32–45, Sept.–Oct. 2001.
- [22] A. Sangiovanni-Vicentelli and G. Martin, "Platform-based design and software design methodology for embedded systems," *IEEE Design and Test of Comput.*, vol. 18, pp. 23–33, Dec. 2001.
- [23] The Ptolemy 2 Project, Univ. California, Berkeley [Online]. Available: <http://ptolemy.eecs.berkeley.edu/>.
- [24] E. A. Lee and A. Sangiovanni-Vicentelli, "A framework for comparing models of computation," *IEEE Trans. Computer-Aided Design*, vol. 17, pp. 1217–1229, Dec. 1998.
- [25] E. A. Lee and Y. Xiong, "System-level types for component-based design," in *Proc. 1st Int. Workshop on Embedded Software*, vol. 2211, Oct. 2001, pp. 237–253.
- [26] V. Sinha, F. Doucet, C. Siska, R. Gupta, S. Liao, and A. Ghosh, "YAML: A tool for hardware design visualization and capture," in *Proc. Int. Symp. Syst. Synthesis*, 2000, pp. 9–14.
- [27] N. Medvidovic and R. N. Taylor, "A classification and comparison framework for software architecture description languages," *IEEE Trans. Software Eng.*, vol. 26, pp. 70–93, Jan. 2000.
- [28] D. Garlan, R. Allen, and J. Ockerbloom, "Architectural mismatch: Why reuse is so hard," *IEEE Software*, vol. 12, pp. 17–26, Nov. 1995.
- [29] H. Tomiyama, A. Halambi, P. Grun, N. Dutt, and A. Nicolau, "Architecture description languages for system-on-chip design," in *Proc. Asia Pacific Conf. Chip Design Lang.*, 1999, pp. 109–116.
- [30] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau, "EXPRESSION: A language for architecture exploration through compiler/simulator retargetability," in *Proc. Design Automation and Test Eur. Conf.*, 1999, p. 100.
- [31] D. C. Luckham, J. J. Kenney, L. M. Augustin, J. Vera, D. Bryan, and W. Mann, "Specification and analysis of system architecture using rapide," *IEEE Trans. Software Eng.*, vol. 21, pp. 336–354, Apr. 1995.
- [32] T. A. Henzinger, B. Horowitz, and C. M. Kirsch, "Giotto: A time-triggered language for embedded programming," in *Proc. 1st Int. Workshop on Embedded Software*, vol. 2211, Oct. 2001, pp. 166–184.
- [33] T. Genssler, O. Nierstrasz, and B. Schoenhage, "Components for embedded software: The Pecos approach," in *Proc. Int. Conf. Compilers, Architecture, and Synthesis for Embedded Syst.*, 2002, pp. 19–26.
- [34] M. M. Gorlick and A. R. R. Razouk, "Using weaves for software construction and analysis," in *Proc. Int. Conf. Software Eng.*, 1991, pp. 23–34.
- [35] C. Szyperski, *Component Software: Beyond Object Oriented Programming*. Reading, MA: Addison-Wesley, 1998.
- [36] E. A. Lee and S. Neuendorffer, "MoML—A Modeling Markup Language in XML," Univ. California, Berkeley, Tech. Rep. UCB/ERL M00/12, Mar. 2000.
- [37] J. K. Ousterhout, "Scripting: Higher-level programming for the 21st Century," *IEEE Computer*, vol. 31, pp. 23–30, Mar. 1998.
- [38] L. Breslau, D. Estrin, K. Fall, J. Heidemann, A. Helmy, P. Huang, S. McCanne, K. Varadhan, Y. Xu, and H. Yu, "Advances in network simulation," *IEEE Computer*, vol. 33, pp. 59–67, May 2000.
- [39] NS: The Network Simulator [Online]. Available: <http://www.isi.edu/nsnam/ns>.
- [40] Simplified Wrapper and Interface Generator (SWIG) [Online]. Available: <http://www.swig.org>.
- [41] P. Chen, D. A. Kirkpatrick, and K. Keutzer. Fast integration of EDA tools and scripting language. presented at Proc. IEEE/DATC Electron. Design Process. Workshop. [Online] Available: <http://www.eda.org/edps/edp01/PAPERS/pinhong.pdf>.
- [42] N. Medvidovic, P. Oreizy, J. E. Robbins, and R. N. Taylor, "Using object-oriented typing to support architectural design in the C2 style," in *Proc. 4th ACM SIGSOFT Symp. Foundations of Software Eng.*, 1996, pp. 24–32.
- [43] Y. Xiong and E. A. Lee, "An extensible type system for component-based design," in *Proc. 6th Int. Conf. Tools and Algorithms for the Construction and Anal. Syst.*, vol. 1785, Apr. 2000, pp. 20–37.
- [44] J. Rehof and T. Mogensen, "Tractable constraints in finite semilattices," in *Proc. 3rd Int. Static Anal. Symp.*, vol. 1145, Sept. 1996, pp. 285–300.
- [45] B. C. Pierce, *Types and Programming Languages*. Cambridge, MA: MIT Press, 2002.
- [46] D. Wetherall and C. J. Lindblad. Extending Tcl for dynamic object-oriented programming. presented at Proc. Tcl/Tk Workshop. [Online] Available: <http://tms-www.lcs.mit.edu/publications/tcltk95.djw.html>.
- [47] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern Oriented Software Architecture: A System of Patterns*. New York: Wiley, 1996.
- [48] T. Groetker, S. Liao, G. Martin, and S. Swan, *System Design with SystemC*. Norwell, MA: Kluwer, 2002.
- [49] Corba middleware software, Object Management Group, Inc. [Online]. Available: <http://www.corba.org>.
- [50] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. San Francisco, CA: Freeman, 1979.
- [51] J. D. Ullman, *Principles of Database and Knowledge-Base Systems: Vol I and II*. Rockville, MD: Computer Science, 1989.



Frederic Doucet (S'98) received the B. Eng. degree in computer engineering from Ecole Polytechnique de Montreal, Montreal, PQ, Canada, in 1999 and the M.S. degree in computer science from the University of California, Irvine, in 2002. He is currently pursuing the Ph.D. degree in computer engineering at the University of California at San Diego, La Jolla.

During his studies, he interned with Lockheed Martin, the Intel Corporation, and Conexant Systems, Inc. He is a Graduate Fellow of the Semiconductor Research Corporation and a Ph.D. scholar of the Fond de Recherche sur la Nature et les Technologies of the Province of Quebec. His research interests are in computer-aided design for system-level design, embedded system design, design automation and software engineering, and management science.

Mr. Doucet is a student member of the ACM.



Sandeep Shukla (M'01–SM'03) received the B.E. degree in computer science and engineering from Jadavpur University, Calcutta, India, in 1991 and the M.S. and Ph.D. degrees in computer science from the State University of New York, Albany, in 1995 and 1997, respectively.

He was a Principal Member of the technical staff of GTE Laboratories (now Verizon) from 1997 to 1999. He was a Senior Formal Verification Engineer and, later, a Staff Component Design Engineer with the Intel Corporation from 1999 to 2001. Prior to joining the Bradley Department of Electrical and Computer Engineering, Virginia Polytechnic Institute and State University in Fall 2002 as an Assistant Professor, he was on the research faculty at the Center for Embedded Computer Systems, University of California, Irvine. Recently, he established the research laboratory Formal Engineering Research with Models, Abstraction, and Transformation (FERMAT) at Virginia Tech to carry out research in applications of formal methods in embedded systems engineering. He has been extensively published in the areas of formal verification, hardware/software co-design, and formal methods for power management.

Prof. Shukla is a recipient of the NSF CAREER Award in 2002. He has co-edited special issues of the *Journal of Circuits, Systems and Computers*, and co-chaired the ACM/IEEE Conference on Formal Models and Methods for Co-Design in 2003. He is a Member of the ACM and Sigma Xi.



Masato Otsuka received the B.E. degree and the M.E. degree in information engineering from Nagoya University, Nagoya, Japan, in 1994 and 1996, respectively.

He has been a Member of Technical Staff with Fujitsu Ltd. since 1996. He has also been a Member of the System Level Design Study Group of JEITA (formerly EIAJ) since 1998. He was a Visiting Researcher at the University of California, Irvine, from 2000 to 2001. His research interests include high-level system design methodologies, models of

computation and communication, and interface specification and verification techniques.



Rajesh Gupta (S'83–M'85–SM'97) received the B.Tech. degree in electrical engineering from the Indian Institute of Technology, Kanpur, India, in 1984, the M.S. degree in electrical engineering and computer science from the University of California (UC), Berkeley, in 1986, and the Ph.D. degree in electrical engineering from Stanford University, Stanford, CA, in 1994.

He is a Professor and Holder of the Qualcomm Endowed Chair in embedded microsystems in the Department of Computer Science and Engineering, UC at San Diego. He was on the faculty of the Computer Science Departments, UC Irvine (UCI) and the University of Illinois, Urbana-Champaign. Prior to that, he was a Circuit Designer with the Intel Corporation, Santa Clara, CA, on a number of processor design teams. He is author and/or coauthor of over a hundred articles on various aspects of embedded systems and design automation and three patents on PLL design, data-path synthesis, and system-on-chip modeling.

Prof. Gupta is a recipient of the Chancellor's Fellow at UCI, the UCI Chancellor's Award for excellence in undergraduate research, the National Science Foundation CAREER Award, two Departmental Achievement Awards, and a Components Research Team Award at Intel. He is Editor-in-Chief of the IEEE DESIGN AND TEST OF COMPUTERS and serves on the editorial boards of IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS and the IEEE TRANSACTIONS ON MOBILE COMPUTING. He is a Distinguished Lecturer for the ACM/SIGDA and the IEEE Circuits and Systems Society.