Efficient System-Level Co-Design Environment using Split Level Programming

Frederic Doucet
Masato Otsuka
Rajesh K. Gupta
Sandeep K. Shukla

ICS
TECHNICAL REPORT

Technical Report # 01-34
(July 1, 2001)

Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425
Information and Computer Science
University of California, Irvine

Technical Report #01-34

Efficient System Level Co-Design Environment using Split Level Programming

Frederic Doucet Masato Otsuka* Rajesh Gupta Sandeep Shukla

Center for Embedded Computer Systems, University of California at Irvine
*FUJITSU Ltd, Japan
E-mail: {doucet,masato,rgupta,skshukla}@ics.uci.edu

Contents

List of Figures

Abstract

IP component based *system-on-chip* design demands an integration, architectural trade-off exploration, and verification environment. Such an environment can only be effective, if system integrators can integrate components from libraries easily, with fast turn-around time, and can efficiently simulate for functional and performance validation.

The advent of C++ based design libraries such as SystemC, Cynlib, and other similar design environments, such as SpecC, allows IP designers to create C++ component libraries, which are either pre-packaged generic elements, or specialized IP blocks. However, mere availability of such components does not necessarily imply ease of integration, or fast architectural exploration. The designers/system integrators need an environment that allows them to think about composable design elements, as hardware elements, and lets them use such elements without having to go through C++ software engineering cycles, such as intergation coding in C++, recompilation, and the corresponding software problems.

Unfortunately, the design and reuse of hardware system level components written in C++, as it exists now, is ad hoc and tedious because of the strong emphasis on inheritance as the basic composition mechanism. Hardware system integrators should be given a better abstraction for composability than having to deal with inheritance, association and other artifacts of object oriented software development. More over, during the architecture exploration phase, *reconfigurability* in existing methodologies for using C++ based hardware design libraries is *static* in the following sense. Every time the component interaction, and architecture is changed, a new *compile-link-test* cycle needs to be initiated. This leads to unnecessarily elongated design time and effort. As a result, we feel a strong need for *dynamic reconfigurability*, without sacrificing the simulation efficiency of compiled objects. However, to achieve this goal, we need to be able to configure component architecture and interconnections in an *interpretive domain*, while simulating at the *compiled code* domain. Also, dynamic runtime configuration requires interaction between the compiled and the interpretive domain at run time. The semantic structure of the compiled objects need to be *reflected* at the interpretive domain for useful interactions at runtime from the interpretive domain to compiled domain, and vice versa. Moreover, C++ being strongly typed, and most interpretive languages being weakly typed or completely untyped, the type in-

formation from compiled objects need to be *exported* to the interpretive domain. At the time of reconfiguration of component architecture and interaction from the interpretive domain, one needs to be able to infer the types of compiled objects at the interpreter run time, and accordingly instantiate compatibly typed objects. This also requires C++ objects in the design library to be type parameterizable.

We solve these problems by 1) creating a interpreted component integration language **CIL** which is based on **Object Tcl**, 2) inventing *type instantiation algorithm* that dynamically assign types, and keeps type compatibility between compiled objects when the object configuration is done from the interpretive domain, 3) allowing *introspection* through a *split level interface*, which is an application of *reflection pattern* from software architectural patterns, 4) creating a special interface definition language (IDL), named **BIDL**, which allows the hardware library designers to define the *exportable interface* of the design library objects, 5) designing a compiler for BIDL, which automatically generates the **split level interface (SLI)** for such objects, and 6) applying the notion of split level programming to the system level design paradigms, such that compiled objects are manipulated from the interpretive domain via the split-level interface, and verification monitors can watch/manipulate, and interact with the activities of the compiled objects from the split level interface.

In this technical report, we elaborate on our framework, called the **BALBOA** *component integration framework* designed for component integration, dynamic reconfiguration, simulation and verification based on any C++ based hardware design libraries. The main focus in this report are on describing the language (CIL) for the integration of a system from IP library components, the interface definition language (BIDL), the dynamic type instantiation algorithm, and concepts, the verification facilities in our framework, and, briefly describe our implementation. We illustrate the concepts using three high level models of a moderately complex design, the AMRM adaptive cache system, using the BALBOA CIL, and BIDL, and the corresponding SystemC based component library.

1 Introduction

The use of C++ class libraries [28] [25] [18] and C++ based languages [8] [20] for hardware and system level modeling is growing steadily. The major advantage of existing methodologies with C++ is that the designers can easily build components that can become a part of intellectual property (IP) libraries. However, design composition with C++ is still tedious and reuse is ad hoc in the *current compile-link-test* methodologies.

One major disadvantage we see in such design paradigms, is that the hardware engineers need to be able to program in C++ for object composition, configure connection between components, and then go through the compilation linking phase which requires them to deal with software engineering artifacts orthogonal to design issues and thus requires extra design time and effort.

Our goal is two fold:

- Relieve the system designer/integrator of the problems of dealing with the artifacts of C++ programming and let them concentrate on design issues,
- Create an environment that allows the designer to dynamically change design configuration, add or delete components, and quickly run simulation to test functionality and performance.

One direct approach could have been using a object oriented scripting language for creating design components and integration. However, simulating non-trivial designs in intrepreted environments would be too inefficient for quick design exploration. On the other hand, if the designer has to use a C++ based library such as SystemC, or Cynlib, they have to learn how to program the glue to connect the components, as well as, have to waste time for every reconfiguration in the compile-link-test cycles.

In the networking research a similar problem was tackled in the VINT [7] project while designing the NS simulator environment. They used a split-programming model to create a network simulation environment, where there are two layers of programming facilities. At the lower layer, they have compiled C++ objects for various network protocol objects, and an the upper layer based on OTcl scripting environment for configuration.

We take a similar approach, albeit, adpated to hardware-software co-design requirements. These requirements are slighly different from the needs of a network simulation environment. In a net-

work simulation environment, building dynamic heterogenous network configuration, and efficient simulation are the main concerns. In the hardware design context, the main concerns are

- creating an abstraction layer to separate concerns about system architecture and software design artifacts,
- creating a rapid design space exploration environment which avoids a compile-link-test cycle of a fully compilation based environment,
- creating an environment that has introspection capabilities to dynamically query types and attributes of design components, and create architecture maintaining type compatibility in the underlying compiled object layer,
- efficient capability to dynamically add test benches and monitors for checking simulation events sequences,
- providing an abstraction from the component library implementation to the designer/system integrator
- providing sufficient efficiency of simulation, comparable to fully compiled design environments
- providing capabilities to mix and match design components of various different abstraction levels, and adding transducers to match the interface behaviors and types, in case of mix and match, and,
- provide visualization, and dynamic control of component interconnection configuration.

In order to resolve all these design forces [9], we created a two layer environment, with a top layer of interpretive domain, and a lower layer of compiled domain. The interpretive domain is implemented as a scripting environment, and the language that is used for achieving component composition, simulation control, test bench creation, and monitoring events, is named **CIL**, or **C***omponent* **I***ntegration* **L***anguage*. The simulation is run at the compiled domain, thereby, achieving the right level of efficiency. The interfacing between the two domains is done through **SLI**s,

or **S***cript* **L***evel* **I***nterfaces*. We provide the designers of component libraries, with an interface definition language, named **BIDL**, or **B***alboa* **I***nterface* **D***efinition* **L***anguage*. A designer of a component, must provide a BIDL interface, for the exportable interface of the compiled objects, which are then automatically compiled using a compiler called the BIDL compiler, to generate the split level interfaces. Once, the SLIs for the components, are available, the upper interpretive layer, communicates with the compiled layer through the SLIs, and there by, abstracting away the implementation details of the compiled components. The type introspection is implemented, so that the component integrator can query the interfaces from the upper interpretive layer, and decide at the integration/composition time, the choice and interconnection architecture.

The test benches can be either compiled objects added to the compiled component library, or can be implemented using CIL scripting from the interpretive layer. Similarly verification monitors can either be objects at the compiled object library, or implemented in CIL. This gives flexibility to system integrators to control configurability, and simulation from the interpretive layer. We have implemented a Graphical User Interface, which can be used in place of scripting, to use point-and-click mechanism for object composition, simulation, and monitoring.

The CIL has type-less scripting to relieve the designer of providing typing details that can be inferred. The type system implements type management during the composition, by using a mechanism called "delayed instantiation" to abstract the typing. Connectivity is also abstracted by using object relationships: the associations, aggregation and composition relationships of a component are manipulatable through the CIL to abstract port-signal-port connection patterns. This infrastructure to compose, instantiate and simulate a design is used at various levels of abstraction.

A few words on Composability: In the current C++ based hardware design libraries inheritance seems to be the prefered object oriented relationship used in reuse methodology, and it tends to be over-used. For example, when designers want to extend a component, they usually derive a new class from a base class and add attributes or behaviors in it. They need to know and understand the hierarchy and the behavior of the base class before that. Because not all classes can be efficiently reused by derivation, they often end up rewriting or replacing the base class by something simpler or more appropriate [19]. Unfortunately, the unconstrained use of inheritance

tend to make reuse difficult. There is no doubt that inheritance is useful, but it should not be the only mechanism used to reuse designs.

To build a complex behavior that cannot be elegantly expressed with only one class, object composition is used. Composition builds a set of objects and relationships implementing a behavior.

To reuse a class in a composition, the class relationships for an object of that class have to be dynamically established (at run-time) to another object with a different functionality, but with a compatible interface.

To reuse a class by extending its behavior, inheritance is used. However, judicious design decisions have to be made in order to choose whether to compose or inherit to build new functionality. To ensuring reuse, there need to be a clear distinction between *how* a design is built, from *what* is built [9]. These are design decisions that have to be captured in design frameworks, like the AMRM composition framework presented in Section 5

Main Contributions: The main contribution of this work is that the CIL provides the designer with the ease of a scripting language, to do design components composition without explicit typing or connection specification. Our system does type inference for proper instantiation, and then takes advantage of SystemC compiled code for simulation efficiency. The impact of this contribution is that the usage of the CIL for design specification enables the use of C++ mainly inside CAD tools for internal design representation. The CIL introduces another abstraction level around the components.

In the jargon of software design patterns, our framework is based on the *reflection* pattern [3], that is built around an introspection capability, for dynamically querying the type information of the underlying design library, and accordingly enable type instantiation from the interpretive layer of the system. Our delayed type instantiation mechanism, allows users to dynamically ask for configuration of system architecture based on objects in the compiled class library at the run time.

Organization of the Report: The outline of this paper is as follows. In Section 2, we give a more detailed discussion of the related work in the area. In Section 3, we introduce the BALBOA component integration environment which consist of the tool environment, the

integration language, the split level interface and the class library for the SLI. Section 4 presents the type system and the delayed instantiation and type propagation subsystems. In Section 5, we present the component integration model of the AMRM adaptive memory system [24] on which we performed communication refinements. In Section 6, we present and discuss the experimental results. Finally, we conclude in Section 7.

We also describe our tool and provide some code listing to show examples of our BIDL, CIL code fragments in the appendix of this technical report.

2 Related Work

Composition is the design activity of assembling small components focused on one task into a more complex component with a richer functionality. Composition is performed at run-time when objects acquire references to other objects. The advantage of composition is that the behavior of a new system will depend on object relationships instead of being defined and hard-coded in one big class. Component technology is emphasized in the software engineering community as a key element [11] in the development of complex software systems. This work draws upon a rich body of work in software engineering and in system level component frameworks.

Design patterns: A design pattern captures a design decision, a class/object architecture to be reused to solve a category of design problems. Design patterns are used to avoid the re-design of system architectures and ensuring that a system can be extended only in specific ways that respect the architecture. Several design patterns exist for efficient and reusable composition of object oriented software [9]. Usually, the composition is by relationships that are set to base types that can be extended by inheritance. For instance, a "creational pattern" is a pattern.

Component Connectivity: Connectivity is a measure of component interfaces. For instance, a component with a large number of signals will be more strongly connected than a component with a smaller number of signals. But, the communication semantic implemented over a relation is also important. For example, a set of signals, that implements the same message passing as a method invocation, can be said more strongly connected. Most hardware or

system level component frameworks have strong connectivity. In SpecC [8], channels with encapsulated protocols or ports with signals are used for communication between behaviors or processing elements. The Colif [4] connections are through ports and nets [22]. Models in NS (Network Simulator [26]) are also strongly connected, because of the nature of the network topologies. For the SystemC and Cynlib C++ class libraries, connectivity is also strong because a discrete event simulation is used to implement RTL VHDL/Verilog semantics. On the other hand, Ptolemy II [23] is less strict because ports can be loosely bound. For all these approaches, inheritance is used to define component types, and port relations/connections are established for the composition. However, class relationships [6] have a very rich semantic for message passing which are powerful connectivity abstractions. In the context of hardware modeling, a relationship between two classes can be successively refined into a handshake over a set of signal connections [12] [16].

Component Integration Strategies: Component integration can be done in the compiled code, graphically, or by scripting. Compiled code integration has been credited as a major factor for the very slow spread of CORBA [13]. It is tedious because many syntactical details that are not necessary for the composition are involved. Graphical integration is easy with the very intuitive block diagram as in VCC [1] and SpecC, but it is difficult to manage for very large designs. Scripting has been used with software component integration for many years. Ousterhout argues that a scripting language for component integration is essential for API abstraction and reuse [15]. Script interfaces for compiled code can be generated using wrapper generators such as SWIG [27] [5], or Tcl Blend [23]. However, wrapper generators present two problems: script syntax is very difficult to generate for complex and parameterized (template) component types [2], and component navigation is impossible because we cannot go inside component hierarchies. On the other hand, NS does not use wrapper generators, but custom scripting interfaces too partially.

Type Systems: Ptolemy has an elaborate type system [14] that statically resolves data types to the most specific type that meets all specified constraints. The type resolution in ML [21] is similar to the type resolution. Colif has a type system to characterize modules (either as software or hardware) and to characterize ports and nets (data and address width

and protocol). Protocol determination can be delayed and solved by propagation in Colif. SystemC has a type system similar to Run Time Type Identification (RTTI), the standard type information checking system in C++, where each SystemC class has a string that can be queried to know its type. But, it is not possible to know template types, because SystemC uses polymorphism in that case.

Communication Refinement: Communication refinement is often discussed in the context of separating computation from communication [17]. In Ptolemy, changes of communication often causes changes of model of computation. In Colif communication refinement is done by using and refining service requests to lower abstractions such as messages (through channels), and port-signal read/write mechanisms. SpecC has a very well defined methodology for communication refinement that involves protocol encapsulation and inlining.

3 Component Integration Environment

The BALBOA design composition environment consists of a component integration language, a split levl interface, C++ and SystemC, and IP libraries. The most basic composition element of the CIL is an object. Objects are composed using one of the relationships shown in Table 1. While objects can be described in any object oriented language, these are turned into reusable components by providing an interface layer to the common C++/SystemC models. Any objects can be used in the framework as a component.

Figure 1 shows the UML use-case diagram for the BALBOA component integration environment. There are two user roles in this system: the system architect uses the component integration language (CIL) to assemble and configure predefined library components; and the library component engineer designs reusable classes using C++ with the SystemC class libraries, and place them in the IP libraries, and generate split level interfaces (SLI) for them using the BIDL language and the BIDL compiler.

3.1 Component Integration Language (CIL)

CIL provides a script-like ease to integrate a system. Table 1 presents a set of object relationship semantics and their syntax implementations in the BALBOA CIL. For most commands, the syntax

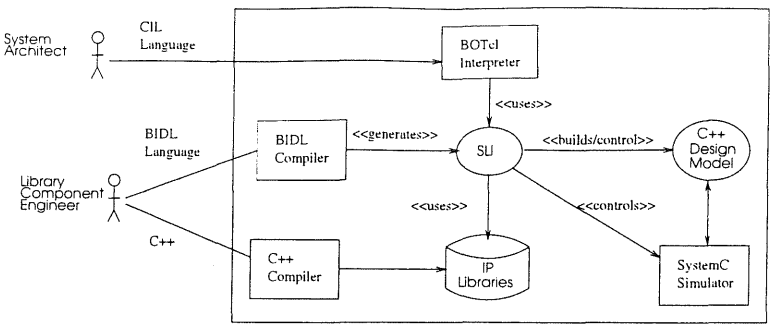

Figure 1. Use-case diagram for the BALBOA component integration environment

is:

```
<entity> <command> <relationship> <value>
```

where the command is applied to the entity. In the association example, the `set_association` command adds to entity `A` an association to entity `B`, named `x`. The syntax for the structural composition command is:

```
<class> <entity name>
```

where `<class>` is the type of the object to instantiate. The composition is specified with the dot (".") operator in the entity name. In the structural composition example, the entity name `A.B` composes (instantiates) an entity `B` inside an entity `A`. The difference between structural and functional composition is that functional composition will invoke the subcomponent behavior, while the structural composition will not necessarily invoke it. Functional composition includes structural composition, but structural composition may not include functional composition. The dot operator is also used to navigate object hierarchies. For example, the `connect` command in Table 1 is applied to entity `B` which is inside of entity `A`. The command will connect the `reset` port of entity `B` to a signal named `sig1`. The name with the dot operator is the full name of an entity with its hierarchy (such as in SystemC). CIL commands also configure and use SystemC for simulation.

3.2 Split Level Interface (SLI)

The SLI is a set of C++ objects that establish the link between the CIL commands and the C++/SystemC models. The OTcl interpreter does not recognize the CIL commands and forwards

Relationship Semantic	CIL Syntax Example
Association	`A set_association x B`
Aggregation	`A set_aggregation y B`
Structural Composition	`Entity A.B`
Functional Composition	`A set_composition z B`
Connection	`A.B connect reset sig1`

Table 1. BALBOA CIL semantics and syntax constructs

the control to the SLI to handle them. We call it a split level interface because it does the link between the interpreted domain of the OTcl interpreter to the compiled domain of the C++ models. The SLI provides a layer around each design component in the C++ design model. From the system architect point of view, the SLI is the design component, because it encapsulates the design component and provides access to only those parts and parameters of the model that enable its use, reuse and adaptation in different design.

3.3 BALBOA Interface Definition Language (BIDL)

The BIDL is a language to export a set of attributes, methods and characteristics of a C++ compiled component to the interpreted hierarchy. A BIDL file is a declarative file, where the class C++ file is pasted and pruned from features that the designer do not want to export. A BIDL file is parsed by the BIDL compiler. This compiler emits the SLI for the input class as C++ output, and also initialization code for some attributes. The BIDL Compiler emit the appropriate code for each component to implement the reflection and some creational design patterns. The BIDL file is used to setup the configuration of these patterns. In other words, the BIDL Compiler will parse a class declaration to emit its SLI. Examples of BIDL files are in the appendix.

3.3.1 SLI Class Library

The SLI class hierarchy is shown in Figure 2. The `SLI` base class is the interface for split level components. All design information used by the BALBOA component integration environment, but not used by the design component behavior should be put in the SLI. When the designer sends a command to the SLI, the command() method is invoked with the parameters provided by the designer. The SLI class is specialized, and the command() method is implemented for specific

SLI behaviors. The `Entity SLI` class is the SLI for all design entities. Design components are aggregated in the `Entity SLI` class. Entities export their relationships to the entity SLI, where they are aggregated in a set. The `Relationship` base class defines the type and the interface for a relationship object and is specialized into the `Association`, `Aggregation`, `Composition` and `Inheritance` sub-classes. The command method accesses this set when it needs to read/set a relationship with the aggregated object. For example, a port-signal connection is captured as a composition relationship with a port object. The SLI will search the `relationships` set for composition with the port name, and if it finds it, it will call the bind method to the port, with the signal argument. The connection command in Table 1 is an example of this procedure. The SLI will search the `reset` composition for the port to bind the `sig1` signal.

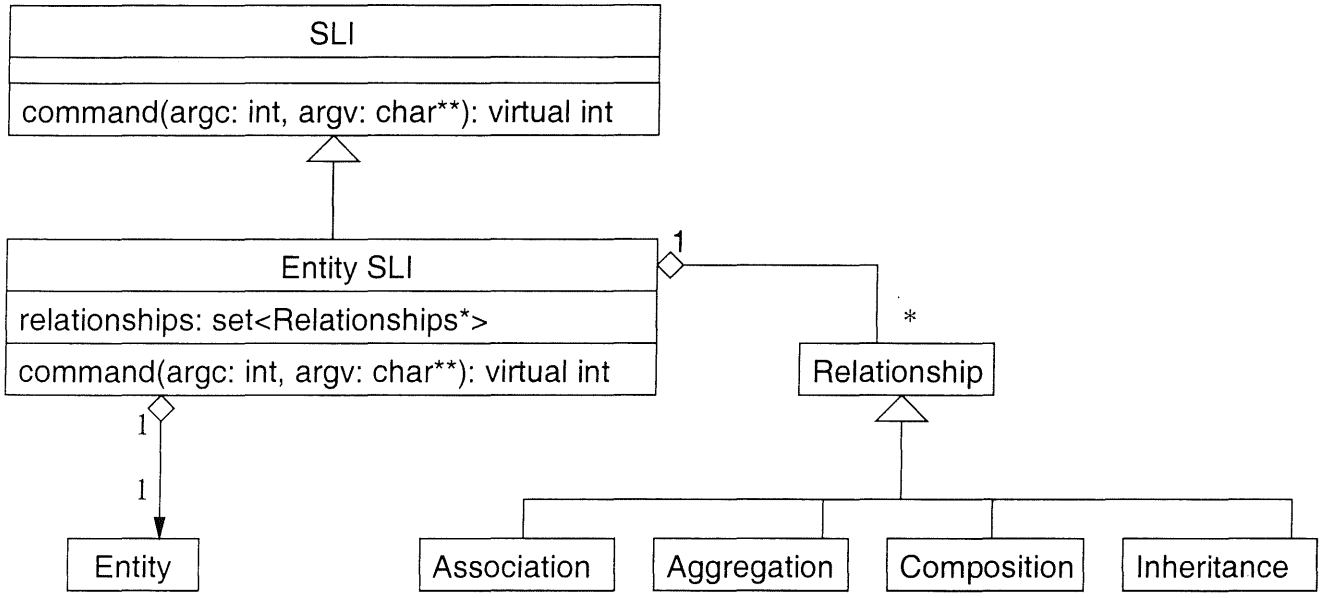

Figure 2. Simplified SLI class hierarchy

4 Type System and Delayed Instantiations

The type system is intended to implement the typing abstraction of the CIL scripting language in the strongly types underlying compiled models. This is subdivided in two tasks: verify if types are compatible when relationships are set; delay a component instantiation when the component is untyped.

4.1 Type Verification

The type verification depends if the type is parameterizable (if a class is templated) or not. If a class is not template, we use RTTI to assert that the types are compatible with the expected types when setting a relationship. If the class is template, it is difficult to verify with RTTI, but there are two possibilities: check only the parameterizable class, or check only the template parameter(s). For example, checking only the parameterizable class occurs when composing an entity with a port: we only need to check that it's a port, not the data format. On the other hand, when connecting two ports together, only the data format needs to be checked. This means only checking the template parameter. We defined a design pattern for template class type checking: every template class will inherit from a base class that is not templated. To type check the parameterizable classes, we compare the type bases with RTTI. To type check the template parameters, we need to capture the type information stored in the class as a string.

The type base class is an abstract interface that is specialized into template classes, each one specific to a real type available in the type system. The instantiation of the template classes is controlled to store the value of the template parameter in the specialized type class as the type of the class. Design template classes will compare these types to assess that the types are right before setting a relationship. Basically, this is an extension to RTTI, which uses almost the same mechanisms as RTTI. But our extension is extendible to multi-templated classes.

4.2 Delayed Instantiations

When an entity is untyped in a CIL script, the BALBOA system delays its instantiation inside the SLI until the type is solved. For example, consider the CIL script 1. Line 1 instantiate an adder, line 3, 4 and 5 instantiate signals. Line 7, 8 and 9 connect the signals to the adder inputs and output. The formats of the data for the adder and the signals are not specified in this script. The parameterizable classes for the component are known: `Adder` and `Signal` C++ classes. However, the templates for the data types are unknown. Thus the component instantiation will be delayed inside the SLI, and all CIL commands issued involving that component will be delayed. A type can be solved by propagation: if an untyped entity is connected to a typed entity, the type of the typed entity is propagated to the untyped entity. Type resolution is basically type propagation

in a component graph, where each untyped component solves a type lattice to determine its own type. If no types are propagated and the design needs to be "closed" for simulation, then the system will allocate the components with a default type. In our current implementation, we use integer as the default type because it is the most convenient type for hardware specification, but a designer can change this type value.

If a class has multiple parameterizable types, then a type lattice needs to be solved before the component can be allocated. This means that all relationships need to be typed before the allocation can happen. A lattice default value is also specified for every component in case the lattice is not solved before simulation.

Script 1 Type-less data composition example

```
1  Adder adder
2
3  Signal sig1
4  Signal sig2
5  Signal sig_result
6
7  adder connect op1 sig1
8  adder connect op2 sig2
9  adder connect out sig_result
```

5 Design Example: AMRM Adaptive Memory System

AMRM is an adaptive cache memory system [24] that can have its properties dynamically changed by software. For example, associativity and line size can be changed by the compiler. The hardware part of the design is a regular cache subsystem, with a modified controller that can execute the extra instructions for cache adaptation.

Figure 3 shows the outline of the procedure we followed for the component integration and communication refinement for the AMRM models. The first step is to integrate and link them with abstract associations in a conceptual model. We implement the concrete message passing semantics and do communication refinement. Figure 4 shows the UML class diagrams and the block diagrams for the component integration and the communication refinements.

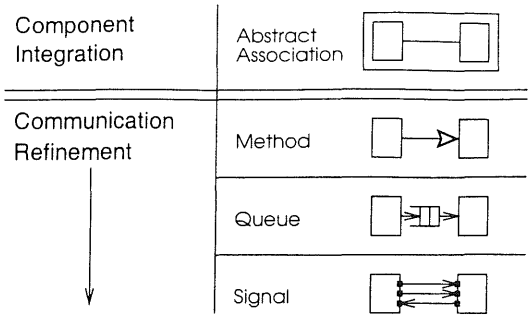

Figure 3. AMRM models in different levels of abstraction

5.1 Component Integration

Script 2 shows the CIL file used at all refinement levels. Line 2 loads the AMRM component library that includes the classes for the cache and memory components and their SLIs to be used in the script. Lines 5 to 7 instantiate two cache controller components named `L1` and `L2`, and a memory controller component named `Mem`. Line 10 instantiates a testbench that aggregates a configurable stimulus list. Line 13 to 15 are OTcl procedure calls that set the associations between the components to enable them to communicate with each other. The refinement process is to re-implement these procedures as the abstract associations are detailed.

Script 2 Component Level Integration

```
1   # Load the AMRM component library
2   load ./libamrm.so
3
4   # Component instantiations
5   Cache  L1
6   Cache  L2
7   Memory Mem
8
9   # Testbench instantiation
10  Testbench CPU
11
12  # Procedure calls to connect components
13  connect_cpu2cache    CPU  L1
14  connect_cache2cache  L1   L2
15  connect_cache2memory L2   Mem
```

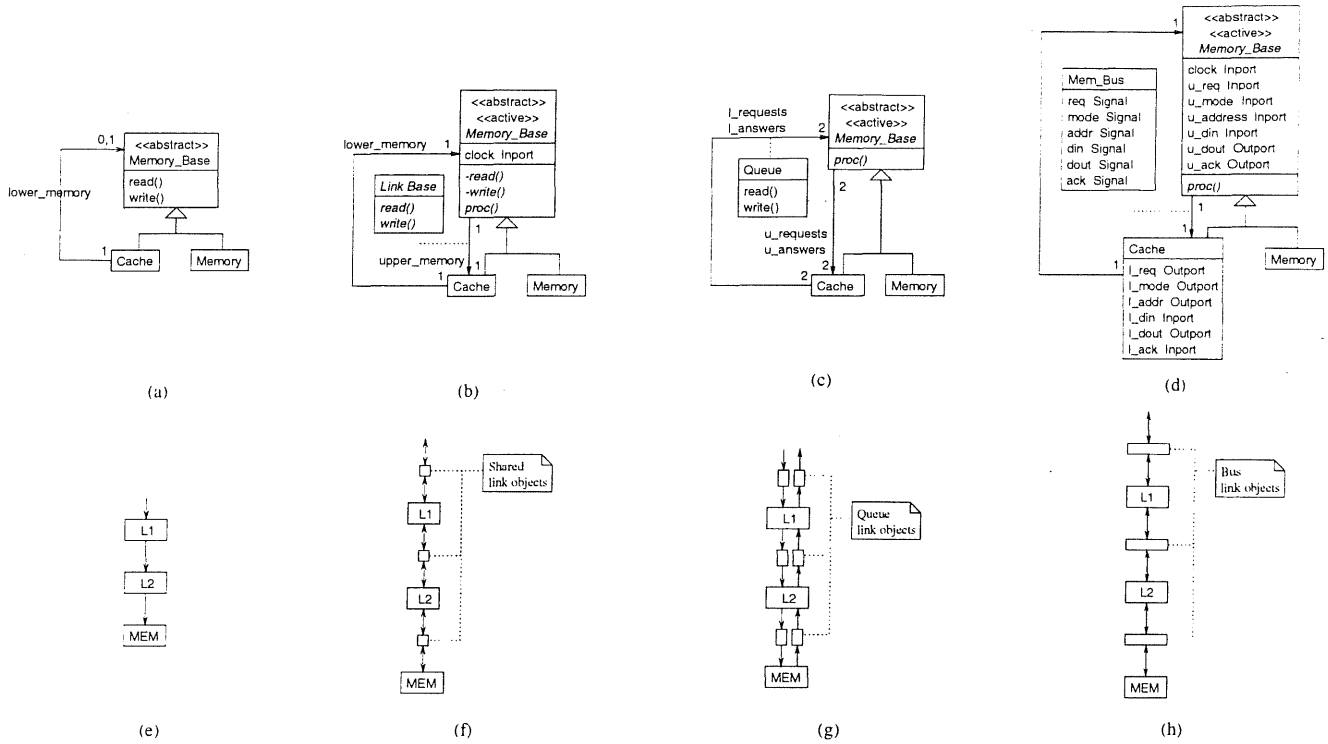

Figure 4. AMRM component integration models with communication refinement: the upper row is for the class diagrams, and the lower row is for the corresponding block diagrams

5.2 Communication Refinement

In the BALBOA system, we perform communication refinement by replacing a communication design pattern by another one with a lower abstraction.

5.2.1 Method Invocations

The first refinement of the abstract associations is to use method invocations to implement the message passing. This is also the model with the highest abstraction that can be simulated. In the class diagram of Figure 4(a), `Memory_Base` is the base class for `Cache` and `Memory` classes. `Memory_Base` has `read` and `write` virtual abstract methods to implement the behavior. These methods are implemented in `Cache` and `Memory` classes. `Cache` class has an association to `Memory_Base` class named `lower_memory`. This association is used to navigate to the lower level of memory. For example, on an `L1` cache read miss, `L1` cache will use this association to call `read` method of `L2` cache. If there is also a read miss for `L2`, `L2` will use its lower memory association

to `read` the data from `Mem`. The block diagram on Figure 4(e), shows how these lower memory associations implement the control flow between the memory levels. In this configuration, two levels of cache are instantiated with one main memory. The Tcl procedure to set the association between two caches is listed in Script 3. This sets the `lower_memory` pointer in the C++ code that was previously exported to the SLI. In this abstract level, the behaviors in the design are sequential. The control flows sequentially from the testbench to the caches, and then to the memory.

Script 3 Cache association through method invocations

```
1  proc connect_cache2cache { U_Cache L_Cache } {
2    $U_Cache set_association lower_memory $L_Cache
3  }
```

5.2.2 Link Classes

Let us now refine further the associations by introducing link classes. Link classes are used to encapsulate a communication protocol through shared structure between the two communicating objects. Figure 4(b), illustrates this conceptual model. The `Link Base` class is introduced to refine the lower memory association. Figure 4(f), shows the block diagram where the link base objects are shared between the components. In this abstraction, we assume that the shared objects are passive: they do not have their own control thread. Therefore, the components need to be active (have their own threads). A reactive process named `proc` is added to `Memory_Base` class. The `proc` process is triggered by an event on `clock` input port and it transitively calls the same private methods as the method invocation model did from `read` and `write` methods. However, because the control goes through that process prior to going to every private method, we need to implement a coarse grain state machine to manage the control flow. A second association named `upper_memory` is added to specify an explicit backward navigation.

5.2.3 Queue Links

Let us refine each association with link class on Figure 4(b) into two associations with the queue shared link objects: one queue for the requests and one queue for the answers. The class dia-

gram on Figure 4(c) illustrates this change. The `l_request` and `l_answer` associations refine the `lower_memory` association, and the `u_request` and `u_answer` associations refine the `upper_memory` association. Script 4 lists the procedure to connect two caches together with queues as link objects. Lines 3 and 4 instantiate the queue components from the BALBOA class library: these components are type-less. Their compiled types will be set according to the types of the associations to which they will connect. Lines 7, 9, 13 and 15 establish the associations between the caches and the queues. Please note that the introduction of abstract data types (ADT) for the request and answer tokens was necessary at this level. These ADTs were not necessary at the method invocation level because the method itself carries the semantic of the request.

Script 4 Cache to cache association with queues

```
1   proc connect_cache2cache { U_Cache L_Cache } {
2     # instantiate queues
3     Queue ${U_Cache}to${L_Cache}_requests_q
4     Queue ${U_Cache}to${L_Cache}_answers_q
5
6     # connect queues to the upper cache
7     ${U_Cache} set_association l_requests \
8         ${U_Cache}to${L_Cache}_requests_q
9     ${U_Cache} set_association l_answers  \
10        ${U_Cache}to${L_Cache}_answers_q
11
12    # connect queues to the lower cache
13    ${L_Cache} set_association u_requests \
14        ${U_Cache}to${L_Cache}_requests_q
15    ${L_Cache} set_association u_answers  \
16        ${U_Cache}to${L_Cache}_answers_q
17  }
```

5.2.4 Signal Links

The lowest level of abstraction in our AMRM models uses signal as the association class. We refine the queue associations into handshake associations. Figure 4(d) shows the class diagram for this model. The `lower_memory` and `upper_memory` associations of Figure 4(b) are still in the design, but their concrete implementations are through the 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 the signal link classes. Figure 4(h) shows the block diagram with the memory hierarchy and the busses. Script 5 lists the procedure to connect two caches through a bus. We use the `connect` directive instead of the `set_association` command, because we bind the concrete signal-port associations. Line 3 instantiates a cache bus named `cb`. Lines 6 to 11 connect the ports of the upper cache to the bus signals, and lines 14 to 19 connect the lower cache to the bus signals. At this level, we remove the abstract data types used at the queue level.

Script 5 Cache to cache association with signals

```
1  proc connect_cache2cache { U_Cache L_Cache } {
2    # instantiate a cache bus
3    Cache_Bus cb
4
5    # connect bus signals to the upper cache
6    ${U_Cache} connect l_req  ${cb}.req
7    ${U_Cache} connect l_mode ${cb}.mode
8    ${U_Cache} connect l_addr ${cb}.addr
9    ${U_Cache} connect l_dout ${cb}.din
10   ${U_Cache} connect l_ack  ${cb}.ack
11   ${U_Cache} connect l_din  ${cb}.dout
12
13   # connect bus signals to the lower cache
14   ${L_Cache} connect u_req  ${cb}.req
15   ${L_Cache} connect u_mode ${cb}.mode
16   ${L_Cache} connect u_addr ${cb}.addr
17   ${L_Cache} connect u_din  ${cb}.din
18   ${L_Cache} connect u_ack  ${cb}.ack
19   ${L_Cache} connect u_dout ${cb}.dout
20 }
```

6 Implementation and Experimental Results

The BALBOA component integration environment is implemented in C++ and SystemC with the packages of the NS simulator for the OTcl extention. We have implemented three AMRM models of different abstract levels in this environment. The statistics of our experimentation are as follows.

The sequential model is composed of 7 classes implementing 1 process using method invocations as the C++ control method. The CIL script size is about 30 lines including set associa-

tion/aggregation commands. This level of abstraction does not use SystemC.

The concurrent model using queues is implemented using 8 classes. The new class is to define an abstract data type to pass in the queues. It has 4 processes and uses queued associations for communication. The script size is about 40 lines including set association/aggregation commands. SystemC starts to be used at this level to capture the process concurrency that was introduced by the shared link objects.

The concurrent model using signals is implemented with 7 classes, 4 processes, and the associations are implemented with the signal-port link pattern. The script size is about 150 lines including signal connection commands.

Finally, the RTL concurrent model [10] is implemented using more that 90 classes. It is composed of 85 processes and the script size is more than 1000 lines including signal connection commands.

As associations are refined, the script sizes grow larger but the number of classes changes very little. This is because the changes of the communication interfaces imply minimal changes on the behaviors. As the abstraction is lowered, the changes are isolated in either the connection procedures in the CIL script, or the well defined message passing interfaces (associations), minimizing the changes to the C++ components. The reason why the number of classes decreases from the queue model to the signal model is that we used an abstract data type (ADT) class to format the data for the cache requests in the queues, but in the signals this ADT is not necessary.

In summary, the size of the design description is not smaller, but the designer works with a much smaller description to make the changes in the total model. The actual model in our case is also as long (or perhaps longer) but it consists of reusable C++/SystemC component code. We introduced another abstraction level that reduces the size of the model but keeps an excellent capability of manipulation, rather than manually changing the underlying SystemC/C++ code.

7 Conclusion

In this paper we presented the BALBOA component integration environment. A design is integrated using a component integration language (CIL) with a split level interface (SLI) that provides an abstraction layer around C++ models. The environment also uses the SystemC

compiled simulator for efficiency. The CIL language abstracts connectivity using object relations, and also abstracts typing. A delayed instantiation type system is used in the SLI to coordinate the underlying strongly typed C++ compiled model. We built models of the AMRM adaptive memory system on which we performed communication refinement design tasks using the BALBOA CIL. We showed how the CIL enables a designer to focus on communication refinement and isolate the changes in CIL procedures with minimum impact on the IP components.

In a broader scope, our work seeks to enhance the ability of hardware system integrators to use C++ based design components in the system integration without worrying about software programming artifacts. We show that C++ based design components can be used in CAD tools for internal design representation. CILs could be used to compose complex designs with a higher abstraction- in other words, to abstract the internal representation and composition mechanisms details of the CAD tool. This was demonstrated by the efficiency of the composition and code reuse in the AMRM models.

Future work includes defining an advanced interface generator to export the SLI API to OTcl, a graphical notation for the CIL, and the refinement of the CIL to include composition into collections (set, vector, bag, etc).

References

[1] M. Baleani, A. Ferrari, A. Sangiovanni-Vincentelli, and C. Turchetti. Hw/sw codesign of an engine management system. In *DATE*, 2000.

[2] D. Berner, D. Jansen, and D. Gajski. Development of a visual refinement and exploration tool for specc. Technical Report TR-01-12, Univ. of Cal., Irvine, 2001.

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

[4] 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 *International Workshop on Rapid System Prototyping*, 2001.

[5] P. Chen, D. A. Kirkpatrick, and K. Keutzer. Fast integration of eda tools and scripting language. In *IEEE/DATC Electronic Design Processes Workshop*, 2001.

[6] B. P. Douglas. *Real-time UML: developing efficient objects for embedded systems.* Addison Wesley, 1998.

[7] L. B. et. al. Advances in network simulation. *IEEE Computer*, May 2000.

[8] D. Gajski, J. Zhu, R. Domer, A. Gerstlauer, and S. Zhao. *SpecC: Specification Language and Methodology.* Kluwer Academic Publishers, 2000.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[10] P. Garg, S. Shukla, and R. Gupta. Efficient usage of concurrency models in an object-oriented co-design framework. In *Design Automation and Test in Europe*, 2001.

[11] J. Hopkins. Component primer. *Commun. ACM*, October 2000.

[12] T. Kuhn, W. Rosenstiel, and U. Kebschull. Description and simulation of hardware/software systems with java. In *Design Automation Conference*, 1999.

[13] G. Larsen. Component-based entreprise frameworks. *Commun. ACM*, October 2000.

[14] E. A. Lee and Y. Xiong. System-level types for component-based design. Technical Report ERL M00/8, Univ. of Cal., Berkeley, Febuary 2000.

[15] J. K. Ousterhout. Scripting: Higher-level programming for the 21st century. *IEEE Computer*, March 1998.

[16] M. Radetzki and W. Nebel. Synthesizing hardware from object-oriented descriptions. In *FDL*, 1999.

[17] J. A. Rowson and A. Sangiovanni-Vincentelli. Interface-based design. In *Design Automation Conference*, 1997.

[18] P. Schaumont, S. Vernalde. L. Rijnders, M. Engels. and I. Bolsens. A programming environment for the design of complex high speed asics. In *Design Automation Conference*, 1998.

[19] A. Taivalsaari. On the notion of inheritance. *ACM Computing Surveys*, 28(3). September 1996.

[20] K. Wakabayashi and T. Okamoto. C-based soc design flow and eda tools: An asic and system vendor perspective. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, December 2000.

[21] Y. Xiong and E. A. Lee. An extensible type system for component-based design. In *6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2000.

[22] S. Yoo, G. Nicolescu, D. Lyonnard, A. Baghdadi, and A. A. Jerraya. A generic wrapper architecture for multi-processor soc cosimulation and design. In *CODES*, 2001.

[23] Ptolemy 2 home page: http://ptolemy.eecs.berkeley.edu/.

[24] Amrm website: http://www.cecs.uci.edu/ amrm.

[25] CynApps home page: http://www.cynapps.com.

[26] Ns: The network simulator home page: http://www.isi.edu/nsnam/ns.

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

[28] SystemC home page: http://www.systemc.org.

A BALBOA Environment and Tool Description

Figures 5 to -8, show the various screen snapshots for the different windows of the BALBOA GUI. The design configuration can be done using the graphical user interface, or through CIL scripts runned through the BALBOA Shell. Figure 5, shows the graphical control panel, which allows user to navigate through the environment, and configure designs, control simulation, visualize the design configuration, and view introspective information about design objects etc.

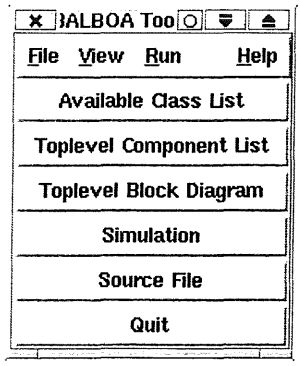

Figure 5. Main Window of Balboa Environment

When the BALBOA shell is started, a number of library are loaded through initialization scripts. They are the following:

1. Core Classes: this library contains all the classes for the basic modeling. It holds the design database and the class for the split level interfaces, the OTcl and the TclCL linkages, the meta model, and the simulator. In the current implementation, the core package is linked with the SystemC simulator.

2. Base Classes: this library contains the classes for the module-clock-signal simulation semantics. It also contains the SLI classes for these basic components. It also holds the linkage code to make sure that when a design class is loaded in the environment it goes add itself to the list of available classes for instantiation.

3. Extra SLI classes: this library provides SLI for basic types.

4. Simulator classes: this library provides the SLI for the simulation control.

5. Testbench and Monitor classes: this library provides the classes to set up design stimulus as a set of signal assignments, or as a set of script commands to be executed. Theses stimulus are setup to be sent at a specified simulation cycle.

6. Behavioral components: this library holds a number of utility classes for design composition. For now, it holds queue classes that we used in the L1 AMRM model.

These library can be found in the sfe2/* and class_lib2/* directories of the source tree.

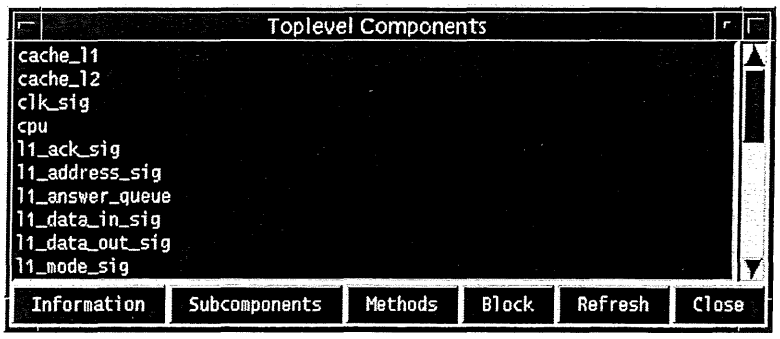

Figure 6. Window to navigate the top level design components

Figure 6 is a screen snapshot of the window which shows the top level design components for the example design of AMRM reconfigurable memory controller. This window is a generic window that can be called in the hierarchy to list subcomponents (by introspection). The buttons at the bottom can call sub-windows to see the information about a component, to list the subcomponents (which instantiate a new instance of this kind of window, but with the list for the inside of the component), the methods that can be applied to a component and the block diagram of a component.

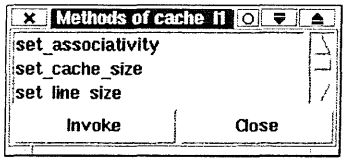

Figure 7. Panel for Introspection: Results of Querying a Class for Available Methods

Figure 7 shows a glimpse of the introspection capabilities, which are needed for dynamic type

and capability information about objects in the design libraries. When selected, and asked for type and attribute and methods of an object from the C++ design library, this window displays those information, which is very useful for dynamic discovery by the user of the design components, and how to interconnect and use them.

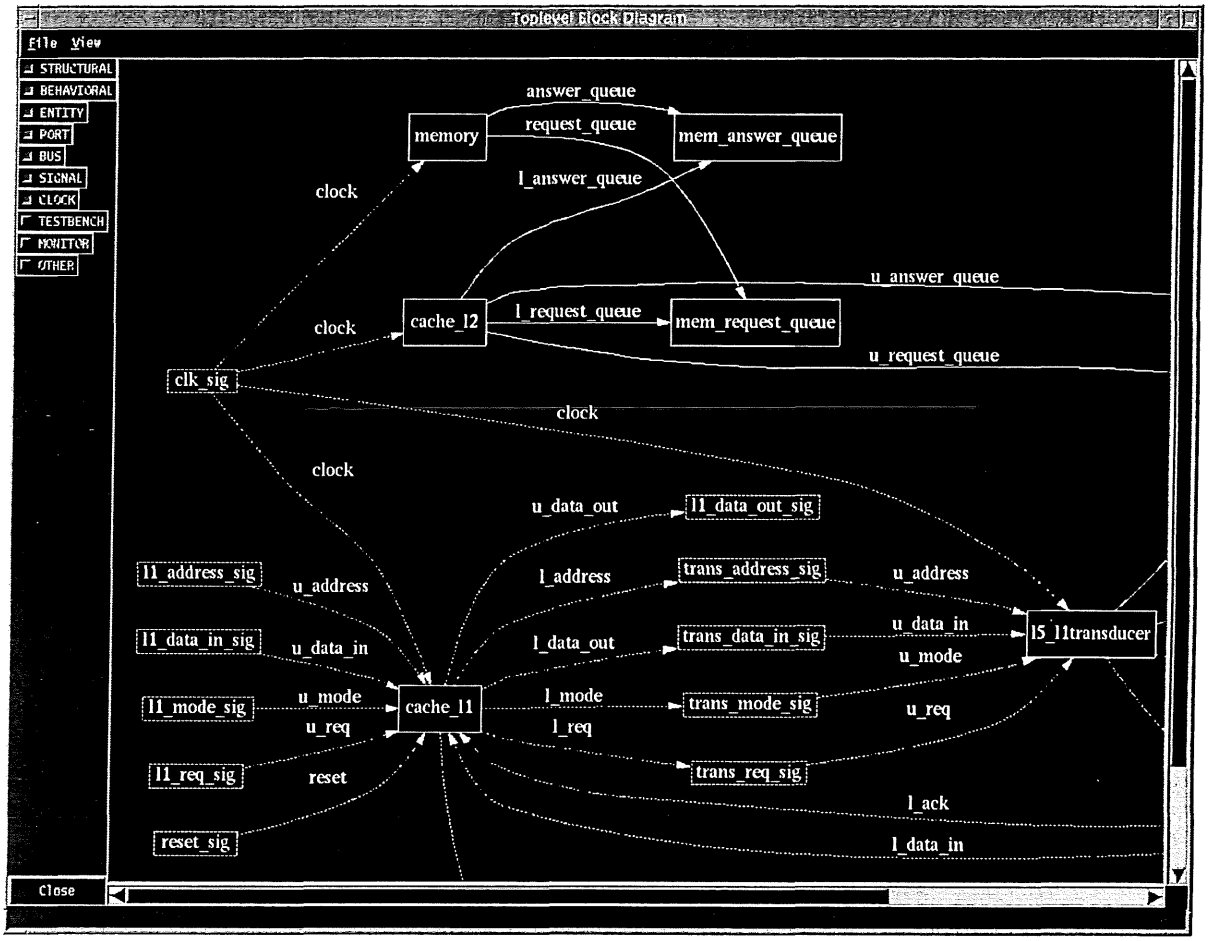

Figure 8. The Block Diagram of The Configuration/Design Architecture Automatic Display Panel

Figure 8 shows the design configuration visually, which automatically updates, as the user enters more design blocks into the configuration. Also, the left hand check box panel in the window, shows that the user can selectively turn on or off display of certain components kinds, for better visualization of some selected components and their interconnections. This window can also be invoked hierarchically (such as the sub-component list window) to view the composition inside of components. This diagram is built by using the introspection provided by the SLI layer.

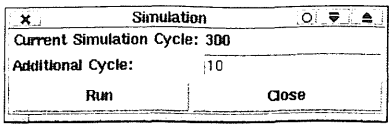

Figure 9. The Simulation Control Window

Figure 9 shows a snapshot of the simulation control window. The current simulation cycle is displayed and the simulation can be runned for a specified number of cycles.

Most of the information displayed in the shown windows are queried by introspection through attributes and methods exported to the SLIs. In the next appendix we show some examples of design, the interface definition for a selected design component (written in BIDL), and the corresponding SLI interface file generated automatically by the BIDL compiler.

B Code Examples

In this appendix, we show code listing for our mixed level AMRM design. In particular, we list an example of a BIDL file, used to provide the exportable interface definition, and also a CIL file, which creates the mixed level design for simulation.

The graphical view of this design composition is illustrated in Figure 8. This AMRM design configuration is composed of two cache memory hierarchy, and one main memory bank. The memory controllers are not at the same level of abstraction: the first one is at Level 5, which means that its communications are through ports-signals-ports style connections, and the second one is of level 1, which means that its communication are through queues. Between the two cache controller sits a transducer that converts signal handshakes into a queue mode communication. On the figure, dotted arrow are signal communications, dotted boxes are signals, and filled arrows are associations (pointer communications) and filled box are design components/entities.

BIDL Listing B.1 shows the description of the interface for the cache controller of level L1. The header description of the class was pasted into the module description, and edited to remove the attributes and methods that were not to be exported. Also, a BEHAVIORAL characterization for this class was added. This is the specification for the "component family" that can be set through the buttons on the left side of the block diagram figure. All attributes and method in the listing

BIDL Listing B.1 Interface File for the L1 Cache Controller

```
1   class Cache_Ctrl_L1 {
2     kind BEHAVIORAL;
3
4     enum replacement_policy_t {RANDOM, LRU};
5     enum write_policy_t {WRITE_THROUGH, WRITE_BACK};
6     enum write_miss_policy_t {WRITE_ALLOCATE, NOWRITE_ALLOCATE};
7
8     Inport<bool> clock;
9
10    // upper queues for data transmission
11    Queue<Queue_Data*>*  u_request_queue;
12    Queue<unsigned int>* u_answer_queue;
13
14    // lower queues for data transmission
15    Queue<Queue_Data*>*  l_request_queue;
16    Queue<unsigned int>* l_answer_queue;
17
18    // counters
19    unsigned int read_counter;
20    unsigned int write_counter;
21    unsigned int read_hit_counter;
22    unsigned int write_hit_counter;
23    unsigned int write_back_counter;
24
25    // configurations
26    bool                    enabled;
27    unsigned int            cache_size;
28    unsigned int            line_size;
29    unsigned int            associativity;
30    replacement_policy_t    replacement_policy;
31    write_policy_t          write_policy;
32    write_miss_policy_t     write_miss_policy;
33
34    // cache array
35    Cache_Array cache_array;
36  };
```

are accessible from the CIL Scripting domain. In this entity, the communications are through queues. Pointers on these queues are set to NULL when the component is instantiated, and then a queue is instantiated in the script and the association to that queue is set by CIL commands.

BIDL Listing B.2 shows the description of the interface for the cache controller of level L5. The header description of the class was pasted into the module description, and edited to remove the attributes and methods that were not to be exported. A STRUCTURAL characterization for this class was added. All attributes and method in the listing are accessible from the CIL Scripting domain.

CIL Script B.1-B.2 do the design setup for the example. The syntax of the commands are different than the ones described in the technical report body because we now use the second implementation of the BALBOA software. However, the semantics are the same. The difference

is that the associations and the ports now have SLIs, so we use them directly to emit the command. In the first version we emitted the commands at the entity level, that used to go to its subcomponent list to perform the command. What happens now is that the command is executed directly in the SLI of the attribute, not in the SLI of the entity. Syntactically, the differences are that the "set_association" command is "link_to", and the port binding ("connect") are through "bind_to". However, the old commands should also be executable because the implementation will be for both of the syntax in the next version.

The CIL Script B.3 sets up the testbench and the monitoring for this AMRM example. Note that these classes accept interpreted stimulus in the simulation.

BIDL Listing B.2 Interface File for the L5 Cache Controller

```
1   Module amrm_l_five {
2     class Cache_Ctrl_L5 {
3       kind STRUCTURAL;
4       enum replacement_policy_t {RANDOM, LRU};
5       enum write_policy_t {WRITE_THROUGH, WRITE_BACK};
6       enum write_miss_policy_t {WRITE_ALLOCATE, NOWRITE_ALLOCATE};
7       Inport<bool> clock;
8       Inport<bool> reset;
9       // upper ports
10      Inport<bool>          u_req;
11      Inport<bool>          u_mode;
12      Inport<unsigned int>  u_address;
13      Inport<unsigned int>  u_data_in;
14      Outport<bool>         u_ack;
15      Outport<unsigned int> u_data_out;
16      // lower ports
17      Outport<bool>         l_req;
18      Outport<bool>         l_mode;
19      Outport<unsigned int> l_address;
20      Outport<unsigned int> l_data_out;
21      Inport<bool>          l_ack;
22      Inport<unsigned int>  l_data_in;
23      // counters
24      unsigned int read_counter;
25      unsigned int write_counter;
26      unsigned int read_hit_counter;
27      unsigned int write_hit_counter;
28      unsigned int write_back_counter;
29      // configurations
30      bool                        enabled;
31      unsigned int                cache_size;
32      unsigned int                line_size;
33      unsigned int                associativity;
34      replacement_policy_t        replacement_policy;
35      write_policy_t              write_policy;
36      write_miss_policy_t         write_miss_policy;
37      // extra commands
38      void set_cache_size(unsigned int arg_cache_size);
39      void set_line_size(unsigned int arg_line_size);
40      void set_associativity(unsigned int arg_associativity);
41      // cache array
42      Cache_Array cache_array;
43    };
44  }
```

CIL Script B.1 Design setup

```
source $env(HOME)/work/src/balboa/designs2/amrm2/amrm.tcl

#
# add stimuli for reading data
#
proc add_read_stimuli {time address} {
  cpu add_signal_stimuli $time l1_req_sig      1
  cpu add_signal_stimuli $time l1_mode_sig     0
  cpu add_signal_stimuli $time l1_address_sig  $address
}
#
# add stimuli for writing data
#
proc add_write_stimuli {time address data} {
  cpu add_signal_stimuli $time l1_req_sig      1
  cpu add_signal_stimuli $time l1_mode_sig     1
  cpu add_signal_stimuli $time l1_address_sig  $address
  cpu add_signal_stimuli $time l1_data_in_sig  $data
}

Clock         clk_sig
Signal        reset_sig
Testbench     cpu

cpu.clk bind_to clk_sig

# signal definitions
Signal l1_req_sig
Signal l1_mode_sig
Signal l1_address_sig
Signal l1_data_in_sig
Signal l1_ack_sig
Signal l1_data_out_sig

# Cache L1 with very low interface abstraction
Cache_Ctrl_L5 cache_l1

cache_l1 set enabled            true
cache_l1 set cache_size         256
cache_l1 set line_size          8
cache_l1 set associativity      1
cache_l1 set replacement_policy LRU
cache_l1 set write_policy       WRITE_THROUGH
cache_l1 set write_miss_policy  NOWRITE_ALLOCATE

#
# connect CPU to cache
# connect signals to the cache
cache_l1.clock      bind_to clk_sig
cache_l1.reset      bind_to reset_sig
# connect to the cpu/tb
cache_l1.u_req      bind_to l1_req_sig
cache_l1.u_mode     bind_to l1_mode_sig
cache_l1.u_address  bind_to l1_address_sig
cache_l1.u_data_in  bind_to l1_data_in_sig
cache_l1.u_ack      bind_to l1_ack_sig
cache_l1.u_data_out bind_to l1_data_out_sig

# Connect to the transducer
ignal trans_req_sig
Signal trans_mode_sig
Signal trans_address_sig
Signal trans_data_in_sig
```

CIL Script B.2 Design Setup

```
64  Signal trans_ack_sig
65  Signal trans_data_out_sig
66  cache_l1.l_req      bind_to trans_req_sig
67  cache_l1.l_mode     bind_to trans_mode_sig
68  cache_l1.l_address  bind_to trans_address_sig
69  cache_l1.l_data_in  bind_to trans_data_out_sig
70  cache_l1.l_ack      bind_to trans_ack_sig
71  cache_l1.l_data_out bind_to trans_data_in_sig
72
73  # Instantiate the transformer (L5 to L1)
74  Transformer l5_l1transducer
75
76  # Connect the transducer to the lower signals
77  l5_l1transducer.clock      bind_to clk_sig
78  l5_l1transducer.u_req      bind_to trans_req_sig
79  l5_l1transducer.u_mode     bind_to trans_mode_sig
80  l5_l1transducer.u_address  bind_to trans_address_sig
81  l5_l1transducer.u_data_in  bind_to trans_data_in_sig
82  l5_l1transducer.u_ack      bind_to trans_ack_sig
83  l5_l1transducer.u_data_out bind_to trans_data_out_sig
84
85  # Instantiate the queues to connect the L2 cache to
86  Queue l1_request_queue
87  Queue l1_answer_queue
88
89  l5_l1transducer.l_request_queue link_to l1_request_queue
90  l5_l1transducer.l_answer_queue link_to l1_answer_queue
91
92  # Cache L2
93  Cache_Ctrl_L1 cache_l2
94
95  cache_l2 set enabled            true
96  cache_l2 set cache_size         256
97  cache_l2 set line_size          8
98  cache_l2 set associativity      2
99  cache_l2 set replacement_policy LRU
100 cache_l2 set write_policy       WRITE_BACK
101 cache_l2 set write_miss_policy  WRITE_ALLOCATE
102 # connect the L2 cache
103 cache_l2.clock bind_to clk_sig
104 cache_l2.u_request_queue link_to l1_request_queue
105 cache_l2.u_answer_queue  link_to l1_answer_queue
106
107 Queue mem_request_queue
108 Queue mem_answer_queue
109
110 cache_l2.l_request_queue link_to mem_request_queue
111 cache_l2.l_answer_queue  link_to mem_answer_queue
112
113 Memory_Ctrl_L1     memory
114 memory.clock bind_to clk_sig
115 memory.request_queue link_to mem_request_queue
116 memory.answer_queue  link_to  mem_answer_queue
117
```

CIL Script B.3 Testbench and Monitor Configurations

```
Monitor m_l1_ack_sig
m_l1_ack_sig.input bind_to l1_ack_sig

#
# stimuli
#
# write data
add_write_stimuli      100 2 3
cpu add_signal_stimuli 110 l1_req_sig 0

cpu add_tcl_stimuli    110 {
  m_l1_ack_sig add_tcl_callback {
    if  {  [cache_l1.cache_array read_valid 0] != "false" ||
           [cache_l1.cache_array read_dirty 0] != "false" } {
      puts "ERROR, test 1 failed for cache_l1"
    }
    if { [ cache_l2.cache_array read_valid 0]   != "true"  ||
         [ cache_l2.cache_array read_dirty 0]   != "false" ||
         [ cache_l2.cache_array read_data  0 2] != 3 } {
      puts "ERROR, test 1 failed for cache_l2"
      puts [ cache_l2.cache_array read_valid 0]
      puts [ cache_l2.cache_array read_dirty 0]
      puts [ cache_l2.cache_array read_data  0 2]

    }
    if { [memory.memory_array  read_data  2] !=  3 } {
      puts "ERROR, test 1 failed for memory"
    }
    m_l1_ack_sig add_tcl_callback {;}
  }
}

# read data
add_read_stimuli       200 2

cpu add_signal_stimuli 220 l1_req_sig 0
cpu add_tcl_stimuli    220 {
  m_l1_ack_sig add_tcl_callback {
    if  {  [ cache_l1.cache_array read_valid 0]   != "true"  ||
           [ cache_l1.cache_array read_dirty 0]   != "false" ||
           [ cache_l1.cache_array read_data  0 2] != 3 } {

      puts "ERROR, test 1 failed for cache_l1"
    }
    if { [ cache_l2.cache_array read_valid 0]   != "true"  ||
         [ cache_l2.cache_array read_dirty 0]   != "false" ||
         [ cache_l2.cache_array read_data  0 2] != 3 } {
    }
    if { [memory.memory_array  read_data  2] !=  3 } {
      puts "ERROR, test 1 failed for memory"
    }
    m_l1_ack_sig add_tcl_callback {;}
  }
}

```

CIL Script B.4 Simulation and Report of Statistics

```
173 simulator run 300
174
175 #
176 # report of counters
177 #
178 set x "cache_l1 read counter      =" ;
179 puts [lappend x [cache_l1 set read_counter]]
180 set x "cache_l1 write counter     =";
181 puts [lappend x [cache_l1 set write_counter]]
182 set x "cache_l1 read_hit_counter  =";
183 puts [lappend x [cache_l1 set read_hit_counter]]
184 set x "cache_l1 write_hit_counter =";
185 puts [lappend x [cache_l1 set write_hit_counter]]
186 set x "cache_l1 write_back_counter=";
187 puts [lappend x [cache_l1 set write_back_counter]]
188
189 set x "cache_l2 read counter      =" ;
190 puts [lappend x [cache_l2 set read_counter]]
191 set x "cache_l2 write counter     =";
192 puts [lappend x [cache_l2 set write_counter]]
193 set x "cache_l2 read_hit_counter  =";
194 puts [lappend x [cache_l2 set read_hit_counter]]
195 set x "cache_l2 write_hit_counter =";
196 puts [lappend x [cache_l2 set write_hit_counter]]
197 set x "cache_l2 write_back_counter=";
198 puts [lappend x [cache_l2 set write_back_counter]]
199
200 set x "memory   read_counter      =";
201 puts [lappend x [memory set read_counter]]
202 set x "memory   write_counter     =";
203 puts [lappend x [memory set write_counter]]
```