An Environment for Dynamic Component Composition for Efficient Co-Design

Frederic Doucet, Sandeep Shukla, Rajesh Gupta
Center For Embedded Computer Systems
University of California at Irvine
{doucet,skshukla,rgupta}@ics.uci.edu

Masato Otsuka
Fujitsu Ltd, Japan
otsuka.masato@jp.fujitsu.com

Abstract

This article describes the Balboa component integration environment that is composed of three parts: a script language interpreter, compiled C++ components, and a set of Split-Level Interfaces to link the interpreted domain to the compiled domain. The environment applies the notion of split-level programming to relieve system engineers of software engineering concerns and to let them focus on system architecture. The script language is a Component Integration Language because it implements a component model with introspection and loose typing capabilities. Component wrappers use split-level interfaces that implement the composition rules, dynamic type determination and type inference algorithms. Using an interface description language compiler automatically generates the split-level interfaces. The contribution of this work is two fold: an active code generation technique, and a three-layer environment that keeps the C++ components intact for reuse. We present an overview of the environment; demonstrate our approach by building three simulation models for an adaptive memory controller, and comment on code generation ratios.

1. Introduction

There is currently a trend of research on using general purpose programming languages in building digital hardware systems, as exemplified by the proliferation of C++ based design languages [21, 19, 26, 16, 13, 23, 8]. 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. A major barrier to adoption of system-level design using C++ is that hardware designers need to understand significant software engineering issues related to components and need to be thoroughly versed in programming abstractions such as inheritance and polymorphism. Often, such concerns are quite orthogonal to the hardware system architectures and design issues, thus actually adding to the time and effort in the system design process. However, there is a definite need to leverage the advantages of programming languages (executable complete system models) in faster and easier system level model construction. In this work, our goal is two fold: relieve the system designer/integrator of the problems of dealing with software engineering and C++ programming issues to enable them to concentrate on hardware system co-design issues, and create a component integration environment that allows the designer to efficiently add or delete components, dynamically change design configuration, and quickly run simulation to test functionality and performance. In this context, we aim at:

1. Creating an abstraction layer to separate concerns about hardware system architecture, software design artifacts and EDA tool integration/setup,
2. Creating a rapid design space exploration environment which avoids a time consuming and programming intensive compile-link-test cycle of a fully compilation based environment,
3. Providing an abstraction from the class library implementation to the designer/system integrator,
4. Creating a component integration environment that has introspection capabilities to dynamically query types and attributes of components, to create architecture maintaining type compatibility in the underlying compiled object layer.

In this paper we present our approach to the construction of executable system models that builds upon C++ class library-based approaches to system modeling. The Balboa [2] composition environment relies on ”smart wrappers” that contain information about the types and object models of the components, and assemble them through an interpreted environment with a ”loose” typing. The paper is organized as follows. The related work is reviewed in Section

Proceedings of the 2002 Design, Automation and Test in Europe Conference and Exhibition (DATE'02)
1530-1591/02 $17.00 © 2002 **IEEE**

2. The key techniques of the Balboa environment presented in Section 3 are layering, component-based design, wrapper generation, scripting based split programming, introspection and typing. Section 4 presents a moderately complex real design implementation in the environment of the Adaptive Memory AMRM [1] system followed by a discussion and interpretation of the results of using this approach.

2. Related Work

This work builds upon ongoing work in the areas of component technology, split-programming and specification language design. A component is a unit of composition be it a function, object, library or a complete program. Component-based design is the activity of assembling small components focused on one task into a more complex component with a richer functionality. Dynamic composition is performed at run-time when objects acquire references to other objects. The advantage of dynamic composition over static composition is that the behavior of a new system will depend on object relationships being defined at runtime, instead of being defined and hard-coded in the files. Component technology is emphasized as a key element [14] in the development of complex software systems. Component integration can be done in three different ways: by programming, graphically, or by scripting.

Programming based composition is tedious because many syntactical details (that are not necessary from an architect viewpoint) must be resolved. Also, many programming decisions independent of the nature of a system model have to be addressed. For example, when building a special adder, a hardware designer should not necessarily be concerned with inheritance, virtual function, friends and other C++ specific constructs, but rather focus on hardware specific characteristics such as bit width, propagation delays, genericity etc. Architecture description languages (ADL) have addressed parts of these problems in the software engineering [18], but for system design, they are often focused on specialized tasks [25], and interoperability can be difficult to achieve [12]. Graphical integration is easy with intuitive block diagrams as in VCC [3], but it is difficult to manage for very large designs. The use of UML has been used also suggested [17] to specify component integration and object interoperability.

The third component integration method is by using scripting. Scripting has been used for many years in software component integration in CAD frameworks. Ousterhout argues that a script language interpreter for component integration is essential for API abstraction and reuse [20]. Script interfaces for compiled code can be generated using wrapper generators such as SWIG [22] [10]. However, current wrapper generator technology presents two problems: script syntax is very difficult to generate for complex and parameterized (template) component types [4], component navigation is impossible because we cannot go inside encapsulated component hierarchies.

Split programming refers to generation of class hierarchies in multiple programming environments with hooks that enable their combined manipulation. NS (network simulator) uses a split-programming model to create a network simulation environment, with two layers of programming facilities: one for building objects, and the other for composing them [6]. A script language is used for network model composition, and the C++ language is used for building components. In NS, a C++ design object used in split-programming has to *inherit* from a class that is visible to the scripting layer. Unlike network simulation, where scripted models are simulated, in hardware system co-design models, simulation efficiency is an important criterion.

System level and hardware specification languages are active areas of research. Most approaches are looking to raise the level of abstraction above the RTL level into either the architectural or the behavioral design space. In system level models, there are many levels of abstraction that are not yet clearly layered, but the understanding is improving [13] [9]. At functional levels of abstraction, models of computation are clear, and in many cases, efficient only for specific application domains. Most of current system level languages are strongly typed, with the exception of the Ptolemy framework that has an elaborate type system [15] that statically resolves data types to the most specific type that meets all specified constraints.

3. The Balboa Composition Environment

In the current C++-based hardware design methodologies, the design components and the simulation kernel are in compiled C++. However, in order to separate the concerns of tool and design setup/control, many design tools use a scripting interface. In the Balboa environment, there is a custom interpreter for a special script language that can execute a number of commands to do component composition, simulation control, test bench creation, and event monitoring. The script language used is named Component Integration Language (CIL). Figure 1 shows the UML use case diagram for the environment. The system architect builds the model of the system using the CIL to direct the composition, by manipulating sets of objects and establishing relationships. The library component engineer builds components in C++, and exports their interfaces to the upper layer by using the Balboa Interface Definition Language (BIDL). The BIDL is basically similar to the CORBA IDL [11], but it has been customized [5] for the needs of composability. The environment is implemented through the following layered architecture:

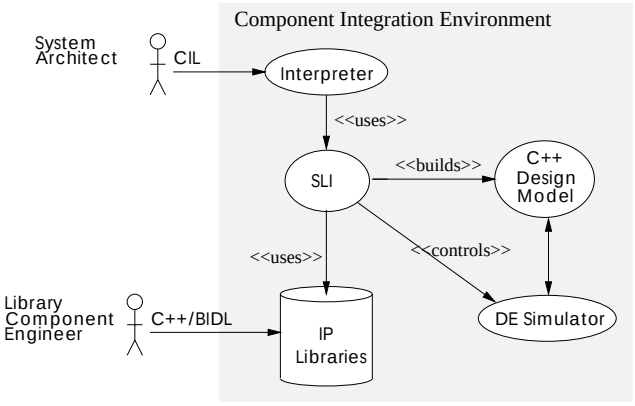

Figure 1. Balboa environment has two user roles: the system architect and the library component engineer

1. **Scripting layer**: Designs are assembled from components configured by using the *Component Integration Language* (CIL).

2. **Wrapper layer**: C++ objects are contained and manipulated inside the component by the *Split-Level Interfaces* (SLI) that implement the split-programming model.

3. **Component library layer**: IP repository where the C++ classes/objects are stored. This layer is independent of the two layers described below. Any compiled C++ object can be placed in this layer.

The design of library component has to be done by a designer who understands C++, the split-level interface generation is automated, and the design composition is done by a system architect with the CIL.

The difference between the previous work and the Balboa environment is the component model driven through a split-programming model through the separation of the component and its split-level interface. Unlike other implementations where a component inherits a class interface implementing the wrapper behavior, in our environment, the SLI wrappers aggregates the design components. This is to have composition and reuse less dependent from the class interface of the internal object, by using object composition and delegation instead of class inheritance. The motivation for the separation of the wrapper from the component is to avoid an inheritance dependency [24]. In other words, reuse by class interface *precedes* the design of the class, while reuse with the BIDL *succeeds* the design of the class. This is because the types are compiled in the type system through the BIDL.

3.1. The Runtime Structure

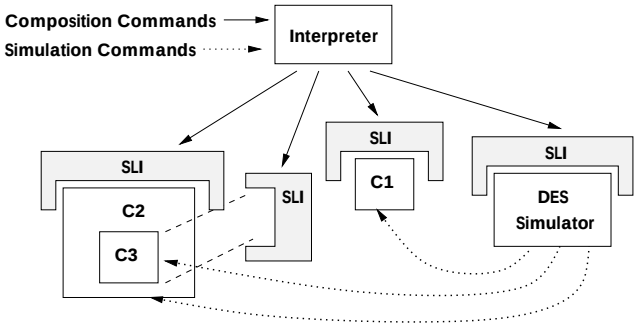

Figure 2. Runtime composition structure of the environment: each design component has a SLI wrapper to process CIL script commands, and runs compiled discrete event simulation

Figure 2 presents a snapshot of the environment at runtime where there are three compiled C++ design components: C1, and C2, which is composed of C3. Each of the components has a split-level interface. There is also a box for a compiled discrete event simulation kernel, such as SystemC simulation kernel, which also has a split-level interface.

3.2. Component Design

Figure 3 illustrates the tool flow and design process for the component designer. The lower part of the figure illustrates the flow for component implementations in C++. The upper part of the figure shows the flow for the component characterization and the exportation of the interface of the components to the interpreted domain. The BIDL compiler generates C++ code to create and configure the split-level interface of a component and to generate the type system information and the specific code for the delayed instantiation and delayed typing. The delayed type instantiation mechanism is explained in Section 3.5. The BIDL compiler also generates the object model configuration specific to the component. The principal steps for using BIDL to export a C++ class to the interpreter are the following: the designer uses the header of the class into the BIDL description and removes the part to be hidden from the interpreted domain. Keywords are also added to configure the generation, such as component families, versioning and template classes handling and specification of available types. From the point of view of system architect, the component and the split-level interface can be the same entity, as shown in the Figure 3 by the dashed rectangle.

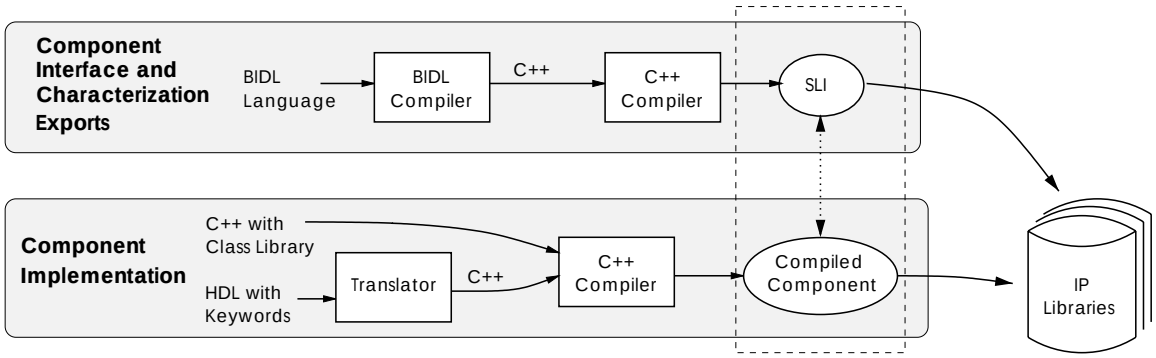

Figure 3. Component design tool flow: write the component in C++, use the BIDL to characterize/export it and generate a SLI

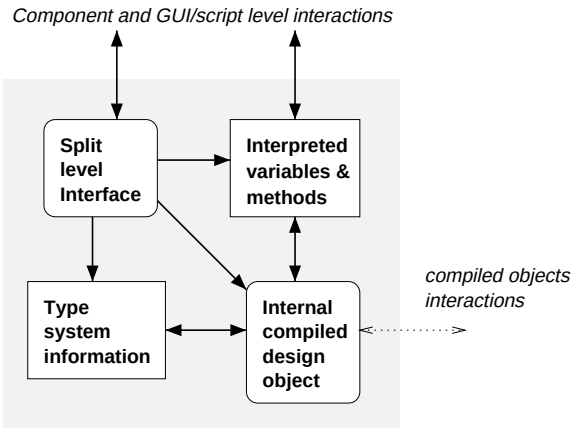

Figure 4. Internal architecture for a Balboa component

Figure 4 illustrates the internal architecture of a Balboa component where there are four blocks: the internal object (for example, such as a SystemC object or any other C++ object or component), the type system information (with an object model), the interpreted attributes/methods (that can be a reflection of the compiled attributes/methods), and the split-level interface routines.

3.3. Interpreted vs. Compiled Domains

As shown in Figure 4, the split-level interfaces are the links of the interpreted domain with the compiled domain. But they also have a state and execute commands. The solid lines on Figure 2 illustrate composition requests from the CIL script language that are only interpreted in the SLIs. However, the simulation commands showed by the dotted lines are delegated to the compiled components. Usually, the simulation control flow is kept only in the compiled layer because interpreted command execution in the SLI can be slower. However, the SLI layer can also interact with the simulation. For example, in our libraries we have a number of stimuli generators and monitors that use the interpreter control flow to compute stimuli and check assertion during the simulation.

3.4. Component Integration Language

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

```
Entity c1
```

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

```
Entity c1.c2
```

The result of this command is the instantiation of an entity named `c2` inside `c1`. The syntax for the composition is the dot "." operator, which is also used to navigate hierarchies.

The CIL also features introspection [7], which is the capacity of an object to query itself to know its structure, attribute and methods. It is similar to self-inspection. For instance, Tcl provides introspection capabilities with the "info" procedure, and Java also provides introspection through the reflection packages. The Balboa environment implement and extend these models to add introspection using a `query` method to the split-level interfaces where the following characteristics of a component can be queried: name, SLI type, C++ type, kind, attributes and methods. For example, the following query:

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

returns the list of attributes for component `c1`. In this case,

there is only the `c2` attribute that is returned as result of the command. This attribute is visible in the interpreted domain, but other attributes might be present in the compiled domain, but not visible if they were not exported. Complex commands can be built to query each subcomponent for information. For example, a command in our environment that query a component for a full characteristics introspection returns the following:

```
Name:  c1
Class:  Entity
Type:  Entity_Imp
Kind:  STRUCTURAL
Attributes:  c2
Methods:
```

Tool behavior for design assembly is built through introspection, looking for attributes or methods, and them introspecting them further to find out the composition possibilities according to an internal object model.

3.5. Type System and Type Resolution

As illustrated in Figure 4, the type system is distributed into all the components. The type system is used to determine the data type for template components. For example, a queue between two components will be specified with no data type, and the system will infer a type according to the data types of the components to which the queue will be connected. Consider the following example in the composition domain:

```
Producer p1
Queue q
Consumer c1
p1.queue_out link_to q
c1.queue_out link_to 1
```

The queue `q` data type is inferred by the type system through introspection of the date type of the `queue_out`. This is subdivided in two tasks: verify if types are compatible when relationships are set; delay a component instantiation when the component is untyped. When an entity is untyped in a script, the system delays its instantiation inside the SLI until the type is solved. For example, when instantiating a port, the data type of the port is unknown. The type system will select which data type to use, and look in a type hash table to find a compiled allocator to instantiate a port for this data type. The allocators are generated by the BIDL compiler. In the current implementation, a type is not guaranteed to be solved if two type incompatible.

4. Experimental Results

In this section, we describe an implementation of a moderately complex system model in the BALBOA environment. The AMRM is an adaptive cache memory system [1] 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.

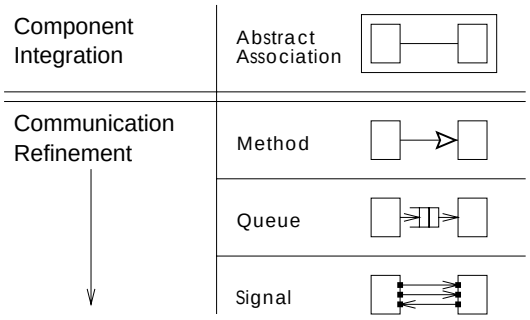

Figure 5. AMRM models in different levels of abstraction

Figure 5 shows the outline of the procedure we followed for the component integration and communication refinement. 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 6 shows the UML class diagrams and the block diagrams for the component integration and the communication refinements. The following script shows the CIL file used at all refinement levels:

```
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
```

Lines 5 to 7 instantiate two cache controller components named `L1` and `L2`, and a memory controller component named `Mem`. Line 10 instantiates a test bench that aggregates a configurable stimuli list. Line 13 to 15 are procedure calls that set the associations between the components for communication. The process is to re-implement these procedures, as the abstract associations are refined.

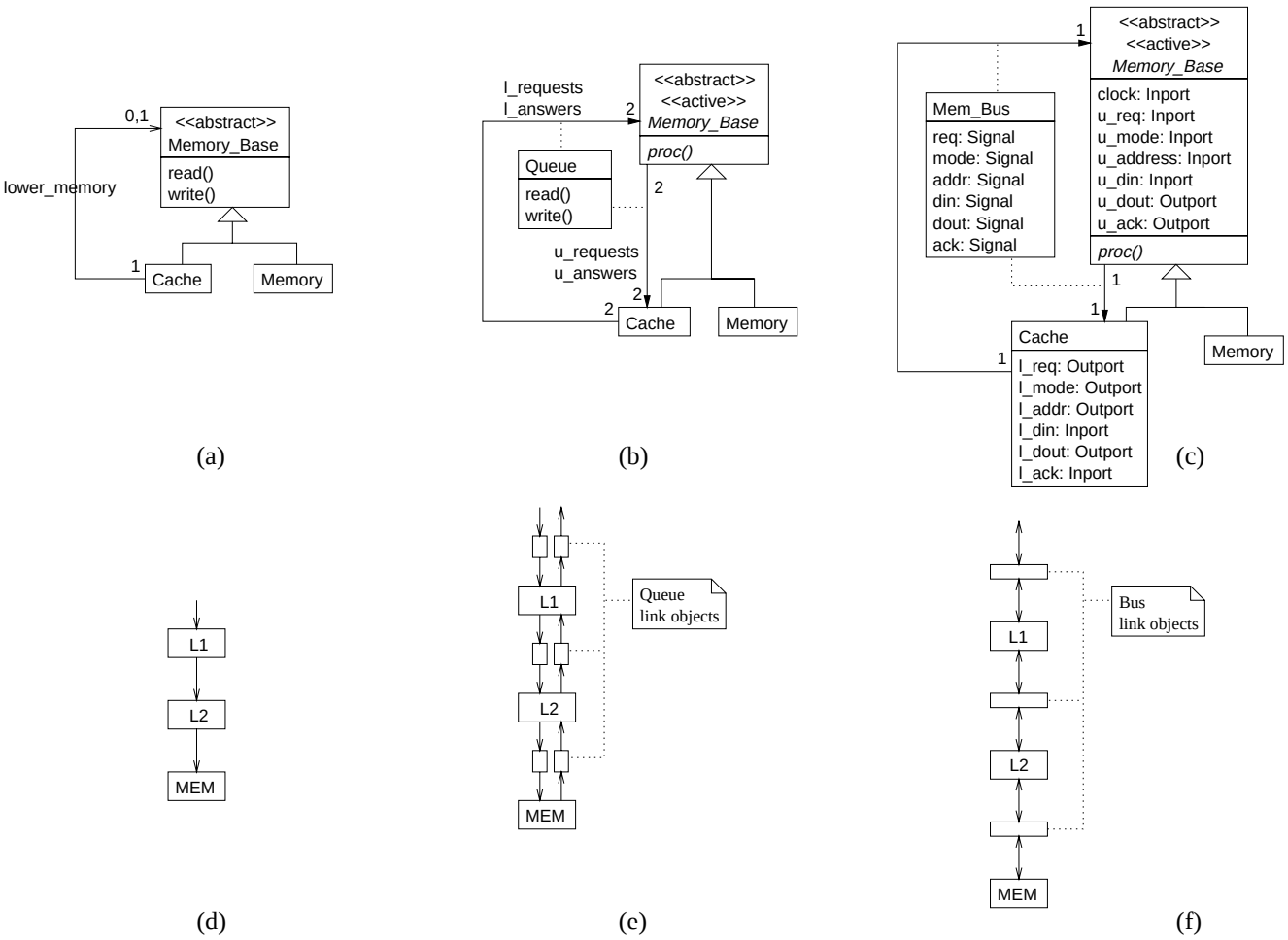

Figure 6. AMRM component integration models with communication refinement: the upper row is for the class diagrams, and the lower row is for the corresponding block diagrams: (a) and (d) refinement 1, (b) and (e) refinement 2, (c) and (f) refinement 3

4.1. First Refinement

The first implementation is to use method invocations to implement the message passing in a sequential model. In the class diagram of Figure 6(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 behavior. The `Cache` class has an association 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 on Figure 6(d), shows how these `lower_memory` associations implement the control flow between the two levels of cache and the main memory. The following procedure sets the association pointers between two caches in structure that was exported to the SLI:

```
1   proc connect_cache2cache { u_c l_c } {
2     $u_c set_association lower_memory $l_c
3   }
```

4.2. Second Refinement

Let us refine the association implemented through method calls into two associations with queues as shared link objects: one queue for the requests and one queue for the answers. The class diagram on Figure 6(b) illustrates this change. This refinement introduces concurrency with the addition of a reactive process named `proc` to the `Memory_Base` class. This process is triggered by an event on `clock` input port and it transitively calls the `read` and

`write` methods. The following script lists the procedure to connect two caches together with queues as link objects:

```
1   proc connect_cache2cache { U_Cache
2                              L_Cache } {
3     # instantiate queues
4     Queue ${U_Cache}to${L_Cache}_requests_q
5     Queue ${U_Cache}to${L_Cache}_answers_q
6
7     # connect queues to upper cache
8     ${U_Cache} set_association l_requests \
9         ${U_Cache}to${L_Cache}_requests_q
10    ${U_Cache} set_association l_answers  \
11        ${U_Cache}to${L_Cache}_answers_q
12
13    # connect queues to the lower cache
14    ${L_Cache} set_association u_requests \
15        ${U_Cache}to${L_Cache}_requests_q
16    ${L_Cache} set_association u_answers  \
17        ${U_Cache}to${L_Cache}_answers_q
18  }
```

Lines 3 and 4 instantiate the queue components (from a class library). The data types of the queues will be set according to the types of the associations to which they are connected. Lines 7, 9, 13 and 15 establish the associations between the caches and the queues. Figure 6(e) shows the architectural view, where each cache level is separated by two queues.

4.3. Third Refinement

The lowest level of abstraction in our AMRM models uses signal handshakes as association link classes. Figure 6(c) shows the class diagram for this model. The queue behavior is still in the design, but implemented 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 the signal objects. Figure 6(f) shows the block diagram with the memory hierarchy and the busses. The following script lists the procedure to connect two caches through a bus:

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

Please note that we use the `bind_to` method of the port object instead of the `set_association` command of the object model. 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.

4.4. Interpretation of Results

Refinement	*# Classes*	*# Script lines*	*# BIDL lines*	*Ratio of IP vs. Generated C++ lines*
1st	7, C++ only	< 30	60	812/809 (1.01)
2nd	8, C++ w/ SystemC	< 40	84	1512/1002 (1.51)
3rd	7, C++ w/ SystemC	< 150	87	1437/880 (1.63)

Table 1. Design statistics of AMRM models: code generation ratio higher at higher abstractions

Table 1 shows the design statistics of the various file sizes and code generation ratio for the 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.

The programming efforts to write BIDL files in this experimentation were non-intrusive 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 vs generated code sizes shown in the rightmost column is increasing as the abstraction is lowered. However, the size of 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 (which is supported by the AMRM library we wrote), except for the type propagation. Work is in progress to investigate what ratios of code generation are obtained in different design context.

5. Summary and Future Work

In this paper, we presented the Balboa composition environment model. It is composed of three layers: an interpreter where a component integration language (CIL) is used to assemble and configure designs; automatically generated split-level interface (SLI) wrappers around each C++ component; and C++ component libraries. The internal architecture of a component implements a type system, an object model, a reflective layer and a split-level interface for a split-programming usage model. The CIL implements introspection and loose typing. This effort is to facilitate the composition activity for the system architect building models with C++ based languages by using code generation and scripting. The usage of the CIL has the advantage of making the code working set much smaller to implement the changes by taking advantage of guided code generation.

The environment features an intermediate language named BIDL to describe exported interfaces of the components and generate the SLIs. Our initial experiments suggest that the usage of the BIDL significantly reduce the programming effort by the component designer. While the total size of the design description is not smaller, the designer works with a much smaller description to make the changes in the full model. As future work, we would like to work further on making the intermediate language independent of the underlying simulation class library by formalizing the simulation semantics in the BIDL.

6 Acknowledgments

We gratefully acknowledge the support for this research from the National Science Foundation (grant CCR-98-06898), the Semiconductor Research Corporation (SRC Graduate Fellowship), the Fond pour la Formation de Chercheurs et l'Aide à la Recherche (FCAR PhD Scholarship) and from DARPA/ITO (contract DABT63-98-C-004).

References

[1] AMRM. Adaptive Memory Platform. Home Page: http://www.cecs.uci.edu/∼amrm.

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

[3] M. Baleani, A. Ferrari, A. Sangiovanni-Vincentelli, and C. Turchetti. HW/SW Codesign of an Engine Management System. In *Proc. Design Automation and Test in Europe Conf.*, 2000.

[4] D. Berner, D. Jansen, and D. Gajski. Development of a Visual Refinement and Exploration Tool for SpecC. Technical Report TR-01-12, UCI, 2001.

[5] A. Borgida and P. Devanbu. Adding more DL to IDL: towards more knowledgeable component inter-operability. In *Proc. Int. Conf on Software Engineering*, 1999.

[6] 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*, May 2000.

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

[8] Cadence Design Systems. VCC. home page: http://www.cadence.com/products/vcc.html.

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

[10] P. Chen, D. A. Kirkpatrick, and K. Keutzer. Fast Integration of EDA Tools and Scripting Language. In *IEEE/DATC Electronic Design Processes Workshop*, 2001.

[11] CORBA website http://www.corba.org.

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

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

[14] J. Hopkins. Component Primer. *Commun. ACM*, Oct. 2000.

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

[16] 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.

[17] G. Martin, L. Lavagno, and J.-L. Guerin. Embedded UML: A Merger of Real-Time UML and Co-Design. In *Proc. Int. Workshop on Hardware/Software Codesign*, 2001.

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

[19] G. D. Michelli. Hardware Synthesis from C/C++ Models. In *Proc. Design Automation and Test in Europe Conf.*, 1999.

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

[21] L. Semeria and A. Ghosh. Methodology for Hardware/Software Co-verification in C/C++. In *Proc. High Level Design Validation and Test Workshop*, 1999.

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

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

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

[25] H. Tomiyama, A. Halambi, P. Grun, N. Dutt, and A. Nicolau. Architecture Description Languages for System-on-Chip Design. In *Asia Pacific Conference on Chip Design Language*, 1999.

[26] C. Weiler, U. Kebschull, and W. Rosenstiel. C++ Base Classes for Specification, Simulation and Partitioning of a Hardware/Software System. In *CS Workshop on VLSI*, 1995.