Introspection in System-Level Language Frameworks: Meta-level vs. Integrated

Frederic Doucet
Univ. of California, Irvine
Irvine, CA 92697 USA
doucet@ics.uci.edu

Sandeep Shukla
Virginia Tech
Blacksburg, VA 24061 USA
shukla@vt.edu

Rajesh Gupta
Univ. of California, San Diego
La Jolla, CA 92093 USA
gupta@cs.ucsd.edu

Abstract

Reflection and automated introspection of a design in system level design frameworks are seen as necessities for the CAD tools to manipulate the designs within the tools. These features are also useful for debuggers, class and object browsers, design analyzers, composition validation, type checking, compatibility checking, etc. However, the central question is whether such features should be integrated into the language, or if we should build frameworks which feature these capabilities in a meta-layer, leaving the system-level language intact. In our recent interactions with designers, we have found differing opinions. Especially in the context of SystemC, the temptation to integrate reflective APIs into the language is great, because C++ is expressive, and already has type introspective packages available. In this paper, we analyze this issue and show that (i) it is a better EDA system architecture to implement reflection/introspection at a meta-layer in a design framework (ii) there are relatively unexplored territories of design automation, such as behavioral typing of component interfaces, corresponding type-theory, and their implication in automating component composition, interface synthesis, and validation, which can be better incorporated if the introspection is implemented at a meta-layer.

1. Introduction

In the recent years system-level design methodologies and languages have been proposed to raise the level of abstraction and promote reuse of intellectual property (IP) libraries. Various approaches to building system models from existing IP components have been proposed, some using programming languages like C++ [12] [15], and some with architecture description languages [3] [1] based on C/C++. SystemC falls in the first category, and rather than using plain C++ for composing components, designers would rather use component composition frameworks, which features automation features such as appropriate component selection, type matching, interface synthesis etc. However, the design structure and behavioral information about the components are often missing in the plain C++ IP blocks, or are lost at the compilation stage: structures are flattened, abstract data structures minimized, and it is not possible to get the precise types of simple components like ports. In other words, IP blocks compiled for fast simulation have reduced design observability. Moreover, programming languages such as C++ are often not suitable to capture many properties of a model, such as temporal behavior, concurrency structure, etc. Such information often needs to be queried by the CAD tools and algorithms. A query about a design object that can be automatically answered by the system is called introspection [2]. Reflection is the architectural technique to allow a component to provide introspective information without manual intervention.

Many HDL simulators implement reflection in entities when they compile a HDL model into a simulatable format. The problem with C++ is that once the compiler builds the executable, the program does not understand the object-oriented structures used during the programming step. For example, a module that displays the status of an object cannot query the object to know its variables and their values. The programmer has to do a customization step before compiling the program, to make sure that the object can display itself. However, it is very cumbersome to manually customize every object in an intellectual property (IP) repository to implement introspective capabilities. If using reflection, the display module could query the object to know what variable it has, and then query the object for their values.

This paper addresses the issues of introspection in system-level language frameworks. We first review the concept of introspection, and how it is implemented in programming languages such as Java. We then discuss various strategies for implementing introspection, which we have tried in the context of the BALBOA [10] framework for component composition. This leads us to our reasoning as to why we converged on an implementation of introspection at a meta-layer over the IP library and implemented

the introspective capabilities by a split-level interface. This allows us to inspect, analyze and manipulate a compiled model with a sophisticated type system. We discuss the advantages and drawbacks of this approach, but more importantly, we give brief insight into how this approach enables composition verification, interface protocol synthesis through reflection of behavioral types.

The contributions of this paper are (1) the proposition of an introspection architecture that automate IP composition by using sophisticated type information about the IP components (2) a comparative study of other ways for automatically collecting information about the types and behaviors, and there by guiding the designers of CAD tools on implementation of introspection by reflection. The BALBOA component framework [10] is a component architecture for system-level modeling with IP design modules built in C++. It can use SystemC designs, as well as other simulation libraries. We describe the experiment in the context of the BALBOA framework.

2. Definitions and Background

Introspection is the capability of a program to explicitly see, understand and modify its own structure [2]. For instance, an introspective object-oriented language allows querying an object for its attributes and methods, their types and values. Introspection is enabled by a *reflection property*, meaning that the program structure is reflected to the program itself. The kind of information describing the structure of the program is called *meta-information* and is commonly used by tools such as debuggers, class browsers, object inspectors, and interpreters.

A programming language is said to be reflective if it provides its programs with meta-information about themselves and a reflection implementation is a software architecture that separates the meta-information from the base program. Among the popular programming languages, C++ carries only very little type information through the virtual function mechanism for polymorphic objects and the supporting run-time type identification (RTTI) mechanisms. In order to have introspection in C++ [5], the following two capabilities are required:

Reification : a data structure to capture the reflected meta-information about the structure and the properties of the program.

Introspection : the capability to query and modify the reified structures.

These concepts are illustrated in the reflection capabilities of the Java programming language where it is possible to query the Java virtual machine (JVM) to know the structure of an object. To such a query, the JVM returns an object that is an instance of a meta class named `Class` that fully describes the type of the object with structures for:

- Inheritance information: references to another object of class `Class` that describe the base class.
- Attributes information: a set of objects of class `Field` that describes all attributes in the class. The `Field` class carries the type information and methods to get or set the value of an attribute.
- Methods information: a set of objects of class `Method` that capture the information of all methods in the class. Such objects carry the type of a return value, parameter description, and can be used to invoke the method.

The reflection API in Java also includes classes for arrays, static variable or methods. Note that part of the virtual machine is reified by a `ClassLoader` class, that "loads in" new classes from the package. This class can be queried to know information about the package and the classes it contains. Most of these methods are to be called at run-time. Because the Java virtual machine understands the structure of the program, it can be queried by a program to know information about itself. In other words, a Java program can understand its own structure at run-time because the run-time infrastructure understands and reifies the structure. Note that there is a security layer to enable or block introspection access to inheritance, attribute and method information, in case secrecy is an issue.

Introspection and reflection are often used in component-based design frameworks. Generally, a component is a stand-alone object with an interface that is not bound to any program. Components are implemented by programmers as units of reuse. They can be as small as a function, an object, a library or as big as a complete program, as long as they have well defined interfaces with explicit external assumptions by the component from the environment.

Component-based architectures [13] consist of components which are objects with a reflected interface, components containers used in component assembly, and scripting used by non-programmers to "wire together" components. System architects have used component frameworks to implement multiple designs successfully. CORBA [7] also has a reflective architecture for service discovery.

3. The Need for Introspection

In the system-level modeling context, various kinds of important information may be reflected. The three main information categories are (i) design information (structural and behavioral), (ii) run-time infrastructure information and (iii) modeling dimension information. This information

when reflected can allow one to design CAD tools to navigate, manipulate, compose and connect components, verify the interface compatibilities and synthesize appropriate interfaces.

Design information can be divided into two subcategories. (i) *Structural design information* is an explicit description of the structure of a design component. This includes the number of processes and their triggering condition, the number of ports and their connections, etc. (ii) *Behavioral information* describes the computation and communication. It includes state machines, the current state, model of computation and the pattern of interaction with its surrounding environment of a component.

Run-time infrastructure information: (i) *Static simulation information* is the topology of an architecture, the number of modules, the number of processes, testbenches, etc. It also includes the description and ordering of simulation steps in the simulation loop. (ii) *Dynamic simulation information* includes the number and types of events in the simulation queue, and in the delta events queue. (iii) *Simulation callbacks* is the possibility to add, query and modify callbacks from the simulation infrastructure to handlers on specific conditions such as component instantiation and deletion, beginning and end of simulation, value changes for signals or attributes and assertions and conditions.

Modeling dimensions information: is used to capture modeling semantics [9]. Information in this category is what classes implement bit-level data types, ports, connectors, signals, channels, processes, structure, and constructs for compositions. Models of computations can also be placed in this category.

4. Introspection Strategies

We consider models built with C++, and a design instance executed by the simulation infrastructure. Figure 1 illustrates such a simple environment. The modeling constructs implement the semantics such as processes, ports and signals through C++ classes. The simulation infrastructure simulates concurrency, bit values and hardware structure. In the ideal scenario, all modeling constructs are introspective in the sense that they provide access to the information enumerated in section 3. However, such introspection capabilities have been missing for a long time in SystemC and other programming language-based modeling frameworks. Recently, a SystemC verification library (SCV) specification [16] described an implementation for data introspection and it usage in verification infrastructure. In this section, we describe the various alternative approaches we experimented within the BALBOA framework to implement introspection on C++ simulation libraries like SystemC.

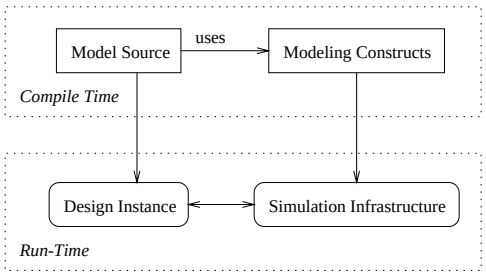

Figure 1. Simple programming language-based modeling environment

The Observer Pattern in Design Components Design components can be modified by adding functions to inspect data structures, adding callbacks on certain events, etc. This is often implemented using the observer design pattern [11]. This approach can be ad hoc and intrusive, and may often break the conceptual structure of the library because the observer pattern does not take into account delta event loops. It is probable that many developers will implement their own extensions to solve the same problem, each in different formats, without communicating, resulting in incompatible extensions, and simulation overheads. It also complicates maintenance when new versions of the simulation infrastructure or of third party tools are released.

Sub-Typing of Modeling Constructs Specialized classes can be derived from the modeling classes in the simulation library to provide introspective methods and reification. This scenario is illustrated in figure 2 where the shaded boxes represent the sub-typed reflective modeling constructs, and requires changes in the model source and instance to use these classes.

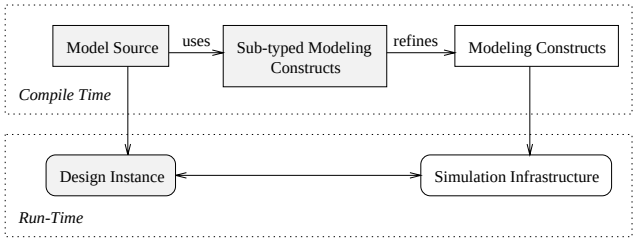

Figure 2. Sub-typing (deriving) modeling constructs to add introspective information

Let us illustrate this with a simplified example of an extension of SystemC input port class to provide access to the precise type of the port, and also to the connection information:

```
template<class T>
class my_sc_in: public sc_in<T> {
public:
  my_sc_in(string exact_type, ...): ... {
    /*store the exact type*/
  };
  sc_channel* bound_to() {
    /* Find and return the
       binding information */
  };
  TypeClass* get_exact_type() {
    /* Return stored exact data type */
  };
};
```

The `my_sc_in` uses the run-time infrastructure to find the binding information, and returns the name of the channel or signal the port is bound to. It also stores the exact type of the port into a variable, which can be queried by the other modules if necessary. Since this approach requires the usage of the specialized classes, they might not be interoperable with other tools because the object with introspective class can have a different memory layout.

Composition Replication for Introspection This scenario requires a separation of the architecture composition code from the component definition code in the model source.

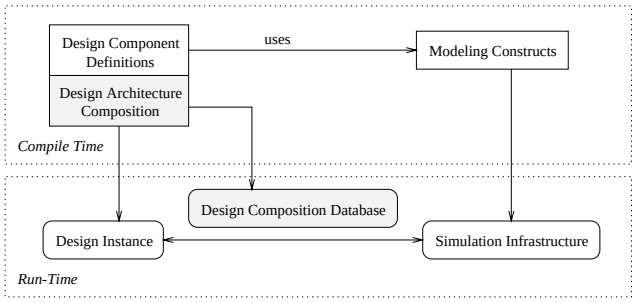

Figure 3. Replication of the composition structure for introspection

Figure 3 illustrates the changes required to the system setup in shaded boxes. A design database is added into the run-time environment to reflect the reified structure to the introspection. Secondly, in the model source, statements are added to store component instantiations and connection bindings into the design database. This information is a reification of the structural design information that is implicit in the design database. This scenario does not require modifications in the IP component definitions nor in the simulation infrastructure, but the addition of composition code of the architecture source.

Using a Declarative Meta-Language The replication of the composition can be encoded in a meta-language like XML instead of being programmed in C++. Component properties can also be described in a meta-layer to reify those properties in the run-time environment.

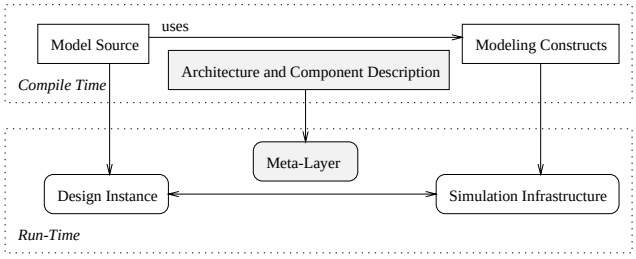

Figure 4. Using a meta-language to configure reification structure with no dependency to the design definition and instance

Figure 4 illustrates the purely descriptive architecture. Design components and the design architecture are described in the meta-language, independently of the modeling source and the reification structures are independently with those descriptions. Using this scenario, no recompilation is necessary to use the meta-layer. But, the XML description has to be in-sync with the compiled description. To do this, a Perl script can be used to translate a C++ class into a XML component description, but it is more difficult to translate an architecture built with an ordered set of C++ imperative statements into a XML description. This approach can be combined with an architectural database using modeling construct subtypes. With that, the design database can generate a meta listing of its architecture to be used by other tools, providing that all necessary information is reified. SCV is a declarative integrated approach, using C macros as a meta-language. Many implementations use XML as the meta-language, as in some flavors of CORBA, MoML [14], Colif [4] and in IP exchange frameworks [8].

5. The BALBOA Component Framework Approach

Figure 5 shows the introspection architecture implemented in the BALBOA component framework environment. This approach does not require the usage of subtypes. It uses composition replication that *requires* that the component definition code be separated from the architecture composition code. This means that components are defined and compiled into IP libraries, as illustrated in the compile-time part of the figure. In the BALBOA implementation, the reification is done through a design database and a type system.

As in the composition replication scenario, the design database captures and manipulates the implicit structure of

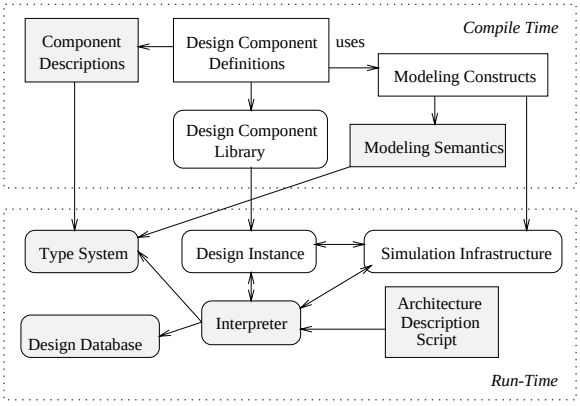

Figure 5. Introspection in the BALBOA component framework

a design. The type system is used to reify the full C++ type information of a component, and its modeling semantics. It captures structural and behavioral information, and non-functional properties of components. It is like the meta-layer in the meta-language scenario, but in the implementation it does active type management.

5.1. Layered Implementation

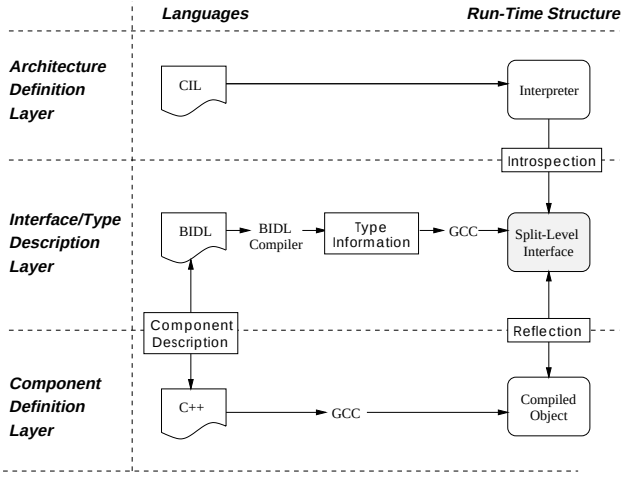

Figure 6. Language and run-time structure layering in the BALBOA component environment

Figure 6 shows an illustration of the language and run-time layers in the BALBOA component framework. Languages as well as the run-time structure are layered, on the left and right side of the figure respectively.

Component Description The BALBOA interface description language (BIDL) is used to describe components to the type system. It was inspired by the CORBA IDL, but it has been extended to support constructs for information listed in section 3. This language is declarative, and is used to specify structural information such as attributes, methods, values, and behavioral types such as state machines and interface automaton. The BIDL is used to generate the reified information as a set of C++ constructs that get compiled into new type system extension structures specific to an IP component.

Architecture Description The Component Integration Language (CIL) is an imperative language used to assemble the components into a design architecture. Basically, that means component instantiation and component connections. This language is based on Object Tcl extensions, thus it is interpreted and loose on typing: components can be partially specified and the CIL interpreter will figure out a way to select a component implementation in C++ that can execute the simulation.

The CIL has introspective capabilities to query its own structure. A subset of the CIL language defines constructs for type manipulation. This subset is used to configure and specify type parameter in components and to assemble new components types by component type composition.

The Split-Level Interface The gray box on figure 6 is the split-level interface (SLI) that manages the link between the interpreted CIL and the compiled C++. The SLI is the run-time structure that controls the component. It serves as a proxy for instantiation, access and behavior invocation. The SLI is the implementation of a part of the type system, specific for a component type. It is also part of the meta-layer that contains the reified information for the component type.

Introspection Through Layering The separation of concerns through between the layers reduces coupling among the reflection mechanisms and the rest of the environment. The type information is declared through the BIDL, and a custom SLI is build for every C++ component class. The introspection is implemented between the CIL and the SLI. The reflection is implemented between the SLI and the compiled C++ object.

5.2 Type Inference and Behavioral Types

In the BALBOA composition framework, one of our main goals is to automate component composition, to construct system models which are correct by construction. The type system was basically implemented for reification. However it created the opportunity for automatic type inference for

component selection. Data type matching has been the first step towards automated component selection and matching it is not enough to ensure correctness. Behaviors of the interfaces may not match (such as communication protocols and model of computation). We need to reflect such behavioral information in the type system, and we do that by capturing behaviors at the interfaces with a behavioral automaton. We use an assume-guarantee [6] type system for type inference and type justification of composition of components. In case the behavioral types do not match, protocol adapters need to be synthesized and we are currently investigating methods to synthesize such adapters.

Such behavioral information is difficult to capture in the components themselves because it often is a meta-property. In the languages based on C++, not only because this would add to the possibility of errors and bugs in the components, but also the expression of such temporal behaviors are cumbersome in C++ and often hidden behind interfaces. As a result in order to be able to automate the behavioral type inference for the interface matching and verification, as well as synthesis of adapters, we use the meta-layer and BIDL to reflect these augmented types.

6. Discussion and Conclusion

After experimenting will all the alternatives, the composition environment evolved into the implementation of the component approach by evolution of the environment. This is justified because this approach minimizes the modifications to the IP modules and the run-time infrastructure, because of the separated reification structure and the expressiveness of the languages to manipulate this structure. This approach is the most promising to reflect all design, runtime and simulation information because the type-system reifies all modeling aspects of components in the environment *plus* their programming implementations. In other words, the type system *is essential* to have exact type reification of the programming constructs, which are the modeling dimensions.

However, this approach has the following disadvantages. The integrity of the information in the meta-layer can be difficult to maintain if there are no callbacks. The interpreter cannot keep track of value changes or event progressions by polling. In other words, if the simulation infrastructure does not provide hooks for callbacks to the type system and the interpreter, it will not be possible to precisely check for certain conditions or assertions without modifying the runtime infrastructure. There is also a performance overhead in terms of execution times because of run-time type checking, and in terms of memory size, because the reified structure can potentially double memory usage.

But the key is to capture the semantics and explore architecture at high-levels of abstraction and thus not let simulation performance dominate the design. In that direction, flexibility is increased because of the separation between the type description, the code generation, the type inference and the base level.

Acknowledgments

This work was supported in part by the Semiconductor Research Corporation under a Graduate Fellowship, the Fonds Quebecois de la Recherche sur la Nature et les Technologies under a Graduate Fellowship, the National Science Foundation, the UC Micro Program and Conexant.

References

[1] R. A. Bergamaschi, S. Bhattacharya, R. Wagner, C. Fellenz, M. Muhlada, F. White, W. R. Lee, and J.-M. Daveau. Automating the design of SoCs using cores. *IEEE Design and Test of Computers*, September-October 2001.

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

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

[4] W. Cesario, G. Nicolescu, L. Gauthier, D. Lyonnard, and A. Jerraya. Colif: A Design Representation for Application-Specific Multiprocessor SOCs. *IEEE Design and Test of Computers*, September-October 2001.

[5] T.-R. Chuang, Y. S. Kuo, and C.-M. Wang. Non-intrusive object introspection in C++: Architecture and application. In *Proc. Int. Conf on Software Engineering*, 1998.

[6] E. Clark, O. Grumburg, and D. Peled. *Model Checking*. MIT Press, 1999.

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

[8] Design and Reuse. IP Repository. Home page: http://www.design-reuse.com.

[9] 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.

[10] F. Doucet, S. Shukla, R. Gupta, and M. Otsuka. An environment for dynamic component composition for efficient co-design. In *Proc. Design Automation and Test in Europe Conf.*, 2002.

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

[12] R. K. Gupta and S. Y. Liao. Using a Programming Language for Digital System Design. *IEEE Design and Test of Computers*, April-June 1997.

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

[14] E. A. Lee and S. Neuendorffer. MoML - A Modeling Markup Language in XML. Technical Report UCB/ERL M00/12, Univ. of California at Berkeley, March 2000.

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

[16] SystemC verification working group. Systemc verification standard specification. Available on the SystemC website, Nov. 2002.