SEPARATING MODELING AND SIMULATION ASPECTS IN HARDWARE/SOFTWARE FRAMEWORK-BASED MODELING LANGUAGES

James Lapalme†, El Mostapha Aboulhamid

Laboratoire LASSO, IR,OUniversité de Montréal
C.P. 6128 , Succ. Centre-Ville, Montréal, Québec, Canada H3C 3J7

Gabriela Nicolescu

École Polytechnique de Montreal, Canada
C.P. 6079, succ. Centre-Ville, Montréal Québec, Canada H3C 3A7

and Frédéric Rousseau

TIMA, 46 av. Felix Viallet, 38031 Grenoble CEDEX, France

الخلاصــة:

من المعروف أنـه كلمـا زاد عدد الترانزستورات في الشريحة الواحدة زادت الفجوة الإنتاجية مع صناعة السليكون اتساعاً. لذلك يسعي العديد من الباحثين لحل المشكلة من زوايا مختلفة. فبينما يرى البعض أن الحل يكمن في لغات النمذجة المختصة بحقل معين يرى آخرون أن الحل يكمن في استخدام لغات النمذجة التي تعتمد على طريقة المكتبة \البنية. ويرى فريق آخر أن استخدام الأدوات المعقدة والمسجلة هو الحل. والحقيقة أن أياً من هذه الحلول ليس حلاً كاملاً. ومع هذا فإن الحلول القائمة على الهياكل المبنية علي التوجـه إلى الهدف مثل نظـام (C) تكتسب زخماً كبيراً وقبولاً من الصناعة. وبالرغم من كل الجهود المبذولة من أجل تطوير هذه الحلول فإن جهوداً قليلة هي التي بُذلت من أجل دراسة تقنيات تصميم البرمجيات اللازمة لتطوير حلول قائمة على الهياكل المبنية. ولذلك فإن الغرض الرئيسي من هذه الورقة هو عرض كيفية استخدام تقنيات هندسة البرمجيات للحصول على حلول أفضل للنمذجة القائمة علي الهياكل. وهذه الحلول تتميز بالفصل الواضح بين الاعتبارات الخاصة بالنمذجة وتلك الخاصة بالمحاكاة. وتقدم الورقة طريقة جديدة للنمذجة الهيكلية تُسمّى (SoCML) و تتمتع بالخاصية السالفة الذكر. وتتمتع (SoCML) أيضاً بمزايا أخرى عديدة مثل التحقق باستخدام الاعتراض، وكذلك الدعم البديل للمحاكاة.

† To whom correspondence should be addressed.
E-mail: james.lapalme@videotron.ca

Paper Received: 16 April 2007; Revised: 15 November 2007; Accepted: 28 November 2007

ABSTRACT

As transistor integration reaches the order of billions, the already significant productivity gap which plagues the silicon industry will only widen further. Many are working on the problem from different angles. Some regard domain-specific modeling languages as a solution. Others believe in modeling languages which are based on a library/framework approach. Yet others believe in sophisticated proprietary tools. None of the current paths seem to be silver bullets. However, object-oriented framework-based solutions, such as SystemC, are gaining a great deal of momentum and acceptance from the industry. Despite all the efforts which have been spent on the development of these types of solutions, very few efforts have been spent on the cornerstone task of investigating which software design techniques and technologies should be used to develop effective framework-based solutions. The main objective of this article is to present how modern software engineering technologies may be used to create better framework-based modeling solutions. These solutions are characterized by a clear separation of concerns between modeling and simulation aspects. A novel modeling framework called SoCML is presented which possesses the above characteristic. SoCML has many benefits such as verification by interception and alternative simulation support.

Key words: logic design hardware description languages, simulation, verification, VHDL, programming languages: design languages, C#, C++, concurrent, simulation and modeling: simulation languages, modeling methodologies, environments

SEPARATING MODELING AND SIMULATION ASPECTS IN HARDWARE/SOFTWARE FRAMEWORK-BASED MODELING LANGUAGES

1. INTRODUCTION

As transistor integration reaches the order of billions [1], the already significant productivity gap which plagues the Electronic Design Automation (EDA) industry will only widen further. Many are working on the problem from different angles:

- Some are working on design flows based on dedicated modeling languages in order to aid designers model complex systems effectively and at higher levels of abstractions then was previously possible [2].
- Others have taken the path of library/framework-based solutions which rely on existing mainstream programming languages. These solutions capitalize on existing tools and technologies and allow the integration of new ones in order to achieve novel design flows [3].
- Others are looking towards sophisticated tools—electronic design automation (EDA)—based on proprietary technology in order to offer "out of the box" design-flow solutions [4].

None of the mainstream approaches seem to be silver bullets; however, object-oriented framework-based solutions such as SystemC [3] are gaining a great deal of momentum and acceptance by the industry. Given this fact, we started, in 2003, working on a new .Net based methodology which enabled the fast and efficient creation of EDA tools for complex systems design. This methodology made the design of a new tool called ESys.Net [5] (Embedded System Design with .Net) possible. Esys.Net: (1) provides most of the concepts of high-level modeling and simulation solutions, (2) inherits features from .Net which allow less error prone modeling, (3) facilitates tool interoperability, (4) permits custom annotation definition, (5) enables model enrichment by annotation (*e.g.* directing synthesis or hooking to verification tools, creating user friendly HDL syntax, *etc*.); and (6) preserves comparative performances with existing modeling and simulation solutions [6,7].

Despite all the efforts which have been made to develop framework-based solutions, as well as the numerous satellite tools, very few efforts have been made on the cornerstone task of investigating which software design techniques and technologies should be used to develop effective solutions.

Software frameworks are quite difficult to build; their design has tremendous impact on:

- Their effectiveness,
- Their ease of use,
- Their ability to promote good designs,
- Their capability to be extended easily.

The software community, over the past decade, has invested a great deal of effort in the domain of software design; pattern-oriented designs are the fruits of these efforts [8]. Moreover, the conflicting needs of the software industry for "rapid time-to-market" (quick design and implementation) solutions which have a low "cost of ownership" (flexible and may easily evolve) has caused the emergence of novel software engineering technologies. The "container-based" implementation approach and 3GL programming languages which support rapid development exemplify these new technologies. Software design principles such as "separation of concerns" have also been maturing, becoming more present in technologies such as software containers, aspect-oriented programming, and strategic programming.

By combining the advanced capabilities of a modern object-oriented programming language such as C#/.Net and the flexibility and elegance of modern software design patterns such as *Inversion of Control* and *Proxy*, it is possible to create a novel framework-based solution for hardware/software system modeling and simulation. We will present a solution which permits a clear and unambiguous separation between the modeling, the verification and the simulation aspects, hence achieving perfect separation of concerns. The level of separation of concerns offered by the solution permits the elaboration and refinement of various simulation engines such as software, distributed and emulation without any modification to those system models which were previously created.

The main objective of this article is to present how new software engineering technologies may be used to create better framework-based modeling solutions. We will (1) present interesting software engineering technologies; (2) show

how current solutions lack "separation of concerns" in their design and discuss the impact; (3) present a novel modeling framework based on the technologies presented earlier; (4) present a simulation framework for the modeling framework; and (5) discuss the benefits of the solution with regards to simulation, synthesis, and verification of the modeled hardware/software systems.

2. BACKGROUND

2.1. Separation of Concerns (SoC)

Probably coined by Edger W. Dijkstra in his paper on the role of scientific thought [9], the concept of separation of concerns is an important principle which lies at the heart of modern software engineering. The basis of SoC is the decomposition of a problem into sub-problems which are orthogonal to each other. SoC takes a classical divide-and-conquer approach to problem solving but relies on aspect decomposition instead of traditional functional decomposition.

Within the context of system modeling and simulation, it can be said that the problem of modeling a system is orthogonal to the problem of simulating a system. Even though both problems are related to one another by common modeling semantics, one is concerned about the "what to simulate" and the other about "the how to simulate". For example, if we have a system such as a cruise control unit which must be modeled and simulated, it should be possible to model the system with certain modeling semantics, such as a hierarchical sea of process, without taking in consideration whether the system will be simulated or emulated.

2.2. C#/.Net 2.0 and Generics

At the end of 2005, Microsoft released the next official versions of .Net and the C# programming language, both versioned 2.0 [10], [11]. Of the many enhancements made to .Net and C#, the implementation of generics types is especially important.

Generic programming, popularized by C++ (templates), is a programming paradigm used by statically type languages in which a piece of software is specified in a way abstracting type information. When a piece of generic software must be used, a programmer must specify a type binding which specializes the software for a given type. Most often, the compiler will duplicate the original generic code but with the type information added in order to enforce static typing. Both Java 1.5 and C++ used this kind of compile time resolution in order to implement the generic programming paradigm.

The designers of .Net took a different approach than the above when implementing generics. The .Net technology is built from the ground up on metadata. In .Net, when a piece of software is compiled, it is transformed into a language agnostic intermediate format called CIL. The CIL instruction-set is based on an abstract stack machine. The intermediate format contains a lot a metadata about the structure of the software that was compiled. The concept of generics was implemented as an extension of the metadata and instruction-set. Because of its implementation strategy, .Net generics are resolved at run-time and not compile time. This makes a big difference at runtime. Through the use of reflection, it is possible to determine if an object is an instance of a generic type as well as the bound types of a generic instance. It is also possible to dynamically bind a generic type and create instance of that binding.

Here is a simple example of dynamic binding and instantiation:

```
Type aType = anObject.GetType();
if (aType.IsGenericType) {
 if (aType.GetGenericTypeDefinition() == typeof(signal<>)) {
   Type signalType = aType.GetGenericArguments()[0];
   Type newType = typeof(GenericSignal<>).MakeGenericType(signalType);
   Object newObj = Activator.CreateInstance(newType));
  }
 }
```

This implementation of generics allows the runtime analysis of generic bindings, the creation of new bindings and the instantiation of those bindings which are very powerful features that we shall explore later in the article. These capabilities, to our knowledge, are unique for a statically type programming environment.

Moreover, the implementation of generics proposed by .Net permits the definition of constraints in order to restrict the types that may be bound to a generic definition.

```
public class Dictionary<K, V> where K : IComparable {}
```

In the above example, a generic dictionary class must be bound to a type for its keys (K) and a type for its values (V). The generic binding is constrained by the fact that the type which is used for the keys must support the IComparable interface.

3. MODELING AND SIMULATION FRAMEWORK

Many efforts have been invested and several contributions have been proposed for system-on-chip modeling and simulation. Current designers have at their disposal efficient solutions for hardware modeling and simulation (*e.g.* VHDL, Verilog); however, few would argue that these solutions are close to being perfect. Most currently available modeling/simulation solutions fall into one of two categories: those which are implemented using a framework based approach and those which are dedicated languages. SystemC [3], ESys.Net [5], JHDL [13], and Ptolemy 2[12] are representative solutions of the first category which may be referred to as domain-specific embedded languages (DSEL) [14]. SpecC [15], VHDL [16], and SystemVerilog [17] are representative solutions of the second category which may be referred to as domain-specific languages (DSL) [14]. In this paper we are concerned with the first category of solutions. The design of a system-oriented DSEL offers an interesting challenge. Since DSELs are implemented using general-purpose programming languages, DSEL designers are constrained by two elements of the host language when defining the necessary system modeling and simulation concepts: (1) syntactical limitations (lexical and grammatical) and (2) a finite set of semantic building blocks.

3.1. Current Solutions

SystemC [3], announced in September 1999, is very popular for system-on-chip design. It is based on a library/framework approach implemented with C++. At its core is an event-driven simulation kernel. SystemC provides all the basic concepts found in HDLs (*e.g.* modules, ports, signals, time, *etc.*). It also provides additional concepts of higher abstraction such as interfaces, communication channels and events.

ESys.Net is a modeling and simulation framework which is based on SystemC. A research team from the Université de Montréal ported the core concepts of SystemC to .Net in order to capitalize on many interesting capabilities of the platform such as threading, reflection and attribute programming.

Ptolemy II is a software framework developed as part of the Ptolemy Project. It is a Java-based component assembly framework with a graphical user interface called Vergil. Vergil itself is a component assembly defined in Ptolemy II. The Ptolemy project studies modeling, simulation, and design of concurrent, real-time, embedded systems. Its focus is on the assembly of concurrent components. The key underlying principle in the project is the use of well-defined models of computation that govern the interactions between components. A major problem area being addressed by the project is the use of heterogeneous mixtures of models of computation.

3.2. Lack of Separation in Current Framework Based Solutions

Traditional HDLs such as Verilog and VHDL were developed from the start with modeling in mind; the simulation of models describe with those languages was a secondary objective. This had a great influence on those standards for there is very little simulation semantics in them. This separation of concerns between modeling and simulation semantics is at the very opposite of environments such as SystemC, ESys.Net, and Ptolemy. We intentionally omit the terminology of "language" to describe these solutions for they are truly simulation solutions and not modeling languages. Our reluctance to qualify the later solutions as modeling languages lies in the fact that there exists no clear boundary between the aspects for simulations and modeling; one cannot model with these solutions and easily change the simulation implementation, especially after the model has been compiled with the simulation framework. In a perfect object-oriented framework, a model should "at all times" be dependent only on the modeling aspects of the framework and not the simulation aspects. The "glue" element between the model and the simulation framework would be the modeling framework which would serve as a contractual interface. ***Figure 1*** represents the dependency architecture of current simulation/modeling frameworks. ***Figure 2*** represents and idealized dependency architecture.

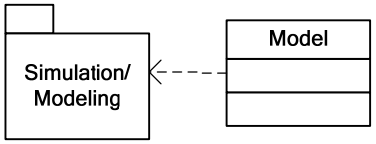


Figure 1. Current frameworks dependencies

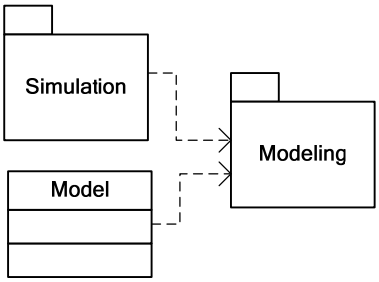


Figure 2. Idealized dependencies

3.2.1. Examples from SystemC and ESys.Net

In SystemC and ESys.Net, a designer has the responsibility to instantiate modeling concepts such as **Event** and **Signal** class instances. The issue here is that since the objects are instantiated by the designer, the implementation of those objects are determined indirectly by the designer. Since the **Event** and **Signal** objects contain code that are directly related to the implementation of the simulation environment, the model created by the designer is indirectly coupled to a simulation implementation. One could argue that it is possible to change the simulation implementation binding of a model changing the implementation to which it is link (such as with Space [18]), but when the model is compiled the binding becomes permanent. As stated earlier, a model should, at all times, only be dependent on the modeling constructs and not the simulation implementation. Since a SystemC and ESys.Net model is directly and indirectly dependant on a simulation implementation, it does not respect true separation of concerns. Another example from SystemC is the concept of a ResquestUpdate in primitive channels. This class method is used to synchronize a primitive channel instance with the delta cycles of a simulation implementation. The concept of a delta cycle should not be present in a pure model, for it has nothing to do with modeling. The only true advantage, with respect to separation of concerns, which ESys.Net has over SystemC is that the simulation infrastructure is not present once models are compiled.

3.2.2. Examples from Ptolemy

In the Ptolemy environment, domains, which are implementations of models of computation, rely on the concept of actors and a director. The actors implement the computation that must take place and the director orchestrates the implementation of the domain. A designer when creating a model must implement and/or assemble predefined actors of the domain. When implementing an actor, the designer must make calls to the director. This implementation style does not respect true separation of concerns for the director has much more to do with simulation then with modeling; hence, there is not true separation between modeling and simulation semantics.

4. SOC MODELING LANGUAGE (SOCML)

4.1. Overview

SoCML is a modeling framework inspired by SystemC and ESys.Net; its implementation only offers a subset of the modeling semantics of the later but a complete implementation could easily be achieved. The main objective of the SoCML project was not the implementation of a modeling framework but rather the demonstration of software framework design techniques which could achieve separation of concerns between system modeling and simulation aspects. SocML contains the concepts of ports, signals, modules, and channels with posses the same semantics as the same named concepts in SystemC and ESys.Net. The major difference is that the semantics are defined only with the aid of abstract classes, virtual empty methods, interfaces, and attributes. Here is the complete list of all the interfaces defined in the modeling framework :

```
public interface Input { }
public interface Output { }
public interface InOut : Input, Output { }

public interface inPort<t> : Input { t Value { get;} }
public interface outPort<t> : Output { t Value { set;} }
public interface inoutPort<t> : inPort<t>, outPort<t> { }

public interface isensitive { inEvent Sensitive { get;} }
public interface ipositive { inEvent Pedge { get;} }
public interface inegative { inEvent Nedge { get;} }

public interface sinPort<t> : inPort<t>, isensitive { }
public interface sinoutPort<t> : sinPort<t>, outPort<t> { }

public interface binPort : sinPort<bool>, ipositive, inegative {}
public interface binoutPort : binPort, outPort<bool> { }

public interface signal<t> : sinoutPort<t> { }
public interface bsignal : binoutPort { }

public interface clock : binPort {}

public interface inEvent {}
public interface outEvent {
   void Cancel();
   void Notify();
   void Notify(long time);
}

public interface biEvent : inEvent, outEvent{}}
public interface var<t> : isensitive { t Value { get;set} }
```

Here is a complete list of the classes defined in the framework:

```
public abstract class BaseModule {
  //Methods with the name Initialize are special to the environment
  //These methods act like constructor and should only contain call to
  //initialize methods of sub-modules and sub-channels


   public virtual void Initialize(){}
   protected virtual void SectionPortBinding() { }
   protected virtual void Wait() {}
   protected virtual void Wait(long time) { }
   protected virtual void Wait(inEvent ev) { }
}

   public abstract class BaseChannel : BaseModule{}
```

The only semantic differences between SoCML and SystemC that are worth noting are:

- the separation of the ***Event*** concept into three sub/super concepts which have directionality;
- a new concept called ***Var*** which represents a variable that may be updated and read in parallel like a signal;
- the SectionPortBinding which is a class method that should contain only port binding code in user defined modules and channels;

- the special treatment of methods found in modules and channels called Initialize. These methods are equivalent to constructors and should only contain method calls to Initialize methods on sub-modules and sub-channels. Initialize methods should only be called once and class constructors should not be used for reasons that will be explained later.
- A top level module which represents a complete model should use a class constructor; however, the constructor must delegate directly and immediately to an Initialize method.

It should be noted that there is no implementation code at all in the modeling framework. This is intentional for as stated earlier: "a modeling framework should only contain modeling semantics".

Here is a simple example of a produce–consume model:

```
public class Consumer : BaseModule {

  public inEvent sync;
  public sinPort<int> input;

  [NonBlockableProcess, Sensitive("input")]
  protected void Consume() {
    Console.WriteLine(input.Value.ToString());
    Wait(sync);
  }
}
```

The model looks very similar to SystemC and ESys.Net. In the consumer module, the **NonBlockableProcess** attribute is equivalent to an SC_METHOD in SystemC. The **Sensitive** attribute indicates that the process is bound to the sensitive event of the signal bound to the port called input. The sensitive event has a clear semantic meaning which is "sensitive to writes on the signals no matter what the value. In the context of a **NonBlockableProcess**, the Wait method call has the same meaning as a next_trigger in SystemC.

In the producer module, the **Blockable** attribute is equivalent to an SC_THREAD in SystemC. The semantics of the producer should be interpreted in the same manner it would be in SysemC.

```
public class Producer : BaseModule {

  public outEvent sync;
  public binPort clock;
  public outPort<int> output;
  private int i = 0;

  [BlockableProcess , Sensitive("clock")]
  protected void Produce() {
    output.Value = ++i;
    Wait(35);
    sync.Notify(25);
    output.Value = ++i;
  }
}
```

```
[ClockDomain(20)]
public class Model : BaseModule {
  private signal<int> sig;
  private clock clk;
  private Producer producer;
  private Consumer consumer;
  private biEvent sync;

  protected override void SectionPortBinding() {
    producer.clock = clk;
    producer.output = sig;
    consumer.input = sig;
    producer.sync = sync;
    consumer.sync = sync;
  }
}
```

Here is an example of a model which used the producer and consumer modules, we can notice the use of the SectionPortBinding method. The **ClockDomain** attribute defines a clock frequence for all clocks defined in its scope. A **ClockDomain** attribute may be assigned to a particular clock in order to override its parent's scope. The objective of this example is not to demonstrate all the possibilities of the modeling framework but to show how semantic found in SystemC and Esys.NET may be defined.

It should be noticed that there are no object instantiations in the model. This information is intentionally left out for two main reasons:

- Object instantiation adds no information to the model. One only has only to imagine that as in C++ the objects are instantiated on the stack and not on the heap because the new keywork is not used.
- By delaying the instantiation of the objects until need (such as at simulation time), we allow the implementation of the semantics to be chosen depending on the context; for example this allows a simulator to instantiate its implementation of the semantic in order to construct the model.

4.2. Design Constraints imposed on SoCML

As mentioned earlier, the main objective of SoCML was the implementation of clear and unambiguous modeling semantics through the use of an object-oriented framework-based approach. In order to keep the framework "clean" of all simulation semantics and artefacts we started with the analysis of ESys.Net in order to determine the implementation elements that had to be eliminated from the framework.

There were two main types of elements which had to be eliminated: method bodies within the framework which where biased towards a certain simulation implementation and framework classes which had to be instantiated which contained code which was biased towards a certain simulation implementation. The two types of elements are clearly exemplified by the code in the Wait methods of the **BaseModule** class and the **Event** class.

The Wait method contains code which pauses the current executing thread in order to switch to the simulation kernel thread. The method also contains code which makes calls to internal methods of the simulation kernel. The **Event** class contains code which permits the scheduling of event instances by making calls to internal methods of the simulation kernel.

One design approach which could be used to loosely couple the core modeling classes from the core simulation classes is the use of a service contract. With this approach, the service contract offered by the simulation core to the modeling core would have been define using an interface type. The modeling core would use the interface type to interact with the simulation core. The only difficulty with this approach is the passing of an implementation of the service contract to the modeling core, however many implementation strategies exist. We did not adopt this approach because it did not fulfill the need to have a complete separation between the two concerns; it only weakened the coupling between both concerns. In order to achieve the required separation, we used only software interfaces to describe all the modeling semantics which we needed. By using interfaces, we eliminated both the problem of instantiation and of biased code fragments. The only semantics which we decided to keep as classes within the framework were Module and

Channel. We made this choice because the use of class inheritance in order to use those semantics permitted a simple and elegant solution

5. A SIMULATOR FOR SOCML

In order to complete the demonstration of our modeling framework design approach, we created a simulation framework for SoCML. In order to achieve our implementation goals, it was necessary to find solutions to the constraints we imposed on the modeling framework. We had to find a way to instantiate implementations for variables contained within a model which where of interface types defined by the modeling framework. We also had to find a way to implement the method bodies of virtual Wait methods in the **BaseModule** class.

5.1. Class Instantiation Problem

The problem of instantiation of an implementation of a variable of a given type is basically the problem of class instantiation. The software design pattern called *Inversion of Control* is a perfect solution for this kind of problem.

5.1.2. Inversion of Control

The software community has a software design pattern which is a solution for a similar problem: *Inversion of Control*.

IoC is a design pattern which enables the decoupling between types [19]. Decoupling is achieved through (1) the use of explicit contract dependency declarations – the term declaration is used here in an implementation agnostic way; (2) the elimination of direct instantiation of a contract implementation by a type instance; and (3) the consummation of a dependency through an interface. Put simply, a type instance designed according to IoC does not instantiate objects which fulfils its dependency needs but rather delegates the instantiation responsibility to an execution environment and consumes the dependencies through interfaces which hide the implementations of the contracts. The execution environment, through the use of a defined dependency need declaration paradigm, locates an implementation for each required dependency of a class instances and instantiates it. Once the implementation instantiated, the environment gives the requester access to it through another defined convention. The environment portion of the pattern is often referred to as a container. IoC is sometimes referred to as Dependency Injection or The Hollywood Principle (Don't call us we'll call you).

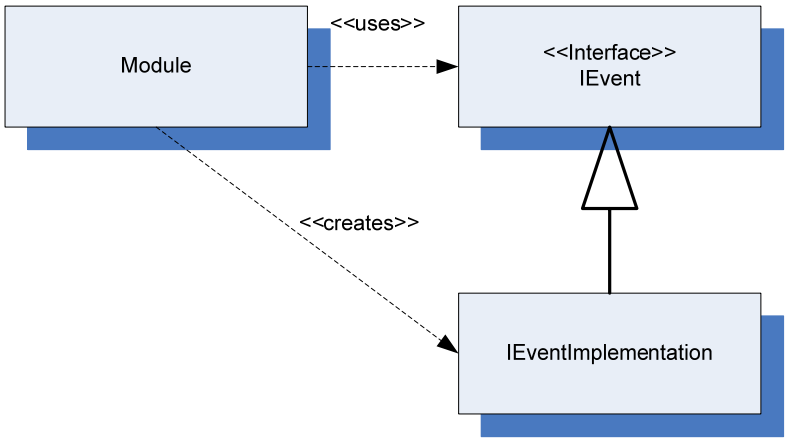


Figure 3. Typical type instantiation

The basic principals of IoC to remember are:

- High level modules should not depend upon low level modules. Both should depend upon abstraction.
- Abstraction should not depend upon details. Details should depend upon abstractions.

Figure 3 represents a typical UML diagram depicting class dependencies between a requestor and a dependency. The **Module** object uses the **IEvent** interface in order to interact with an **Event** object but is must also instated the implementation of the **IEvent** interface. The **Module** object is the requestor and the **Event** object is the dependency. In most current modeling solutions (ESys.Net and SystemC), the **IEvent** interface does not exist but in substituted for the implicit interface of their respective event classes.

Figure 4 represents a UML diagram depicting a modified version of Figure 3 using the IoC pattern.

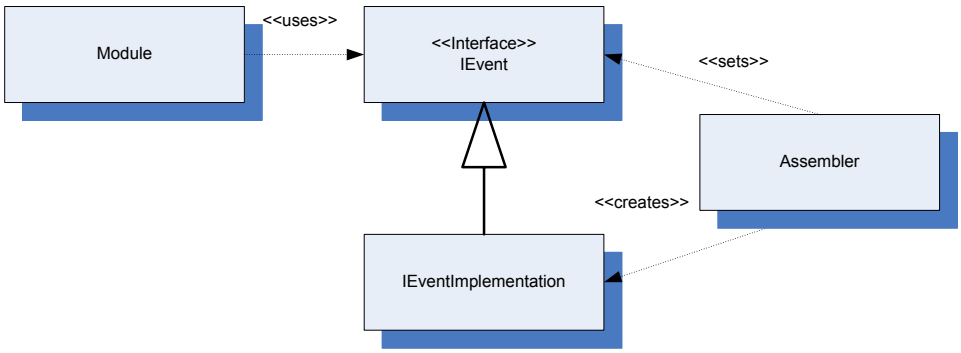


Figure 4. Inversion of control

In the modified diagram, the **Module** object no longer has the responsibility of instantiating the implementation of the interface. The instantiation task is delegated to an **Assembler** object.

5.1.3. Typical Implementation Techniques

Three mainstream techniques exist in order to implementation the declaration of dependencies and the passing of dependency instances between a dependency requestor and a container: contextualized lookup, constructor injection and setter injection. The three techniques use the concept of a container which is an execution environment for the requestor which acts as the **Assembler**.

Contextualized Lookup

Contextualized Lookup is a technique in which the container makes dependencies available via an interface/method that the requestor implements to indicate that is has dependencies. The interface/method, which is implemented by the requester, typically receives a reference to a lookup service.

```
public class Module : Serviceable{
    IEvent event;
    public void service(ServiceManager sm) {
        event = (event) sm.lookup("IEvent");
    }
}
```

The above implementation uses a lookup service which instantiates the **IEvent** object for the **Module**; dependency information is often stored in a configuration file for the lookup service to use.

Constructor Injection

Constructor Injection is a technique in which the container makes dependencies available to a requestor via a class constructor. Dependency declaration information is often retrieved via reflection on the constructor. The container is responsible for instantiating objects and passing dependency implementations.

```
public class Module {
    private IEvent event;
    public Module (IEvent event) {
        this. event = event;
    }
 }
```

Setter Injection

Setter Injection is a technique in which the container makes dependencies available via setter methods after instantiation.

```
public class Module {
   private IEvent event;


/**
     * @service name="IEvent"
     */
   public void setIEvent(IEvent event) {
       this. event = event;
   }
}
```

By its very nature IoC involves loss coupling between service requesters and services providers. This loss coupling promotes easier code maintenance, easier code reuse, and a lot of flexibility.

5.1.4. Inversion of Control With Reflection

The mainstream techniques used to implement IoC are usually satisfactory in the context of business applications, but they are not sufficiently transparent to be used in the context of a modeling framework. It would be necessary to "pollute" the modeling framework with IoC implementation mechanisms which have nothing to do with modeling.

If we come back to the IoC design pattern, the basis of the pattern is to create a contract with the aid of an interface, which permits a service requestor to declare the need of a service whose implementation will be chosen by a container. This pattern applies very well to the model/modeling/simulation tuplet :

- the model may be seen as the requestor;
- the modeling framework may be seen as the service interface definitions;
- the simulator may be seen has the container;
- the interface declaration in the model may be seen as a service requests.

Through the used of reflection [5, 20] , a simulator can dynamically discover the interface declarations and understand them as service requests.

Our SoCML simulation frameworks uses model analysis through reflection in order to act has a container. The core of the analysis is very similar to the one used in ESys.Net, the only significant difference is the instantiation upon detection of the modeling semantic declarations. The analysis and implementation strategy used by the simulator of the simulation framework is made possible by the runtime resolution of generic in the .Net framework. The strategy could not have been used by an implementation in Java or C++.

Here is a fragment of the pseudo-code used by the core model building algorithm of our simulator.

```
BuildModel(BaseModule module) {
      mduleType = GetType(module);
      fieldDefinitions = GetFields(moduleType);
  foreach   field in fieldDefinitions {
    fieldInstance = GetInstance(field,module);
    fieldType = GetType(field)
    If(fieldInstance is unset){
    Select(fieldType){
       Case inEvent,outEvent,biEvent :
         FieldInstance = new EventImplementation;
       Case clock :
         fieldInstance = new ClockImplementation;
       Case signal :
         boundType = GetGenericBoundType(fieldType)
         customeSignalType =
            CreateSignalType(boundType,ImplSignalType)
         fieldInstance = new customeSignalType
       Case module
         proxy = CreateProxy(fieldInstance)
         fieldInstance = proxy;
    }
  }
  Foreach subModule in module BuildModel(subModule)
}
```

We can clearly see the use of the IoC design pattern in the above code fragment. We may view the model as the service requestor and the simulator as the container. The handing over of the service is done in an alternative way from the ones presented earlier. The model declares its need of a service by declaring a variable of a type defined in the modeling framework. Through introspection of the model, the simulator finds the service declarations and sets them with an instance of an appropriate implementation. The simplicity and elegancy of the solution is made possible by the reflective and dynamic generic capabilities of .Net.

5.2. Virtual Method Implementation Problem

The problem of implementing a virtual method without the consumer being aware often arises in the context of distributed applications. In traditional distributed applications, consumers use an object which impersonates a remote object. The responsibility of the impersonating object is to offer a simple interface — usually an interface which is identical to the remote object — to the consumer and marshal the calls to the remote object. The same basic technique can be use for a virtual method implementation problem. The technique is based on the proxy design pattern.

5.2.1. Proxy Design Pattern [8]

The proxy design pattern is one the structural patterns defined by the GoF. The GoF defines structural patterns as:

"Structural patterns are concerned with how classes and objects are composed to form larger structures. Structural class patterns use inheritance to compose interfaces or implementations. As a simple example, consider how multiple inheritances mixes two or more classes into one. The result is a class that combines the properties of its parent classes. This pattern is particularly useful for making independently developed class libraries work together."

The intent of a proxy is to provide a surrogate or placeholder for another object to control access to it. There are various flavors of proxies depending on their usages such as:

- remote, where you represent a remote object through a local object;
- virtual, which provides on demand creation of expensive objects;
- protection, which controls access to the original object;

- smart reference, also known as a smart pointer, which provides "decorated" functionality to the proxied object (such as a smart pointer, persisted object loader, or wrapper object for multithreaded operations to a single-threaded object.)

Here is the UML diagram of the pattern:

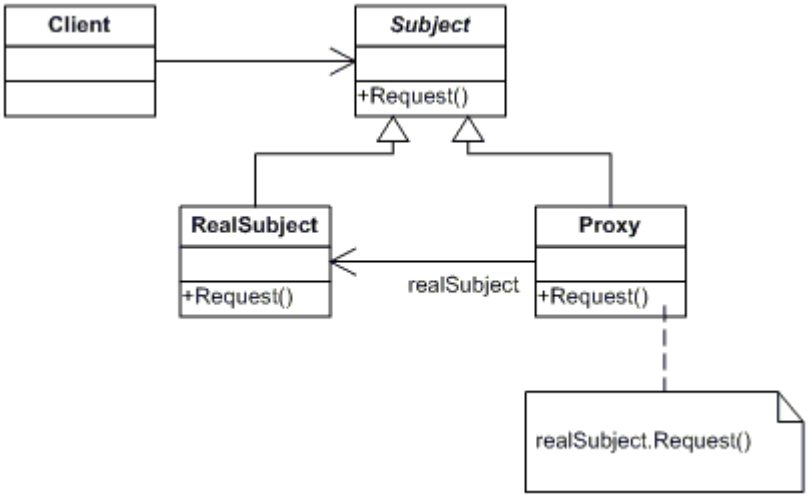


Figure 5. Proxy pattern

The proxy maintains a reference that lets the proxy access the real subject. The proxy may refer to a Subject if the RealSubject and Subject interfaces are the same. It also provides an interface identical to Subject's so that a proxy can by substituted for real subject and controls access to the real subject and may be responsible for creating and deleting it. The proxy may have other responsibilities depending on the kind of proxy.

The Subject defines the common interface for RealSubject and Proxy so that a Proxy can be used anywhere a RealSubject is expected.

RealSubject defines the real object that the proxy represents.

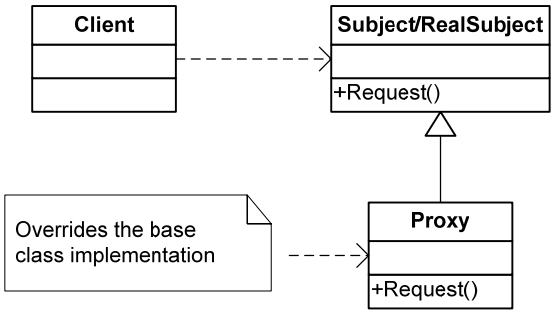
5.2.2. Combining IoC and Proxy Design Pattern

In traditional distribution applications, the developer is aware that he is using a proxy object for he usually either requests an instance of the proxy from a factory style software layer or he must explicitly instantiate it. Both of these approaches were not suitable for the implementation of the simulation framework for we did not want the system modeler to have to be aware of the underlying mechanics of the simulator in order to achieve separation of concerns.

In order to solve the problem, we used a combination of the IoC and proxy design patterns. In the simulation implementation that we propose, the implementation of the Wait methods of a **Module** or **Channel** are achieve by the use of a proxy which catches the calls and delegates them to the simulator. The interception of the calls is achieved by:

- Creating dynamically a derive type from a user-defined module/channel type at runtime.
- Overriding all the virtual Wait methods with an implementation which delegates the call to directly/indirectly the simulator.

In our implementation, the Subject and RealSubject are both the same type; a user defined module or channel type. To be more precise, we can state that the Subject is the public interface of a user-defined module/channel and the RealSubject is the implementation of that public interface. In order to obtain a type which derives from the Subject we must derive from the user-defined type.



We achieve the creation of the proxy class by using the DynamicProxy.Net framework distributed by the Castle project [21]. The framework supports the creation of proxy type from generic types.

```
proxyGenerator.CreateClassProxy(type, new MyInterceptor(this)));
```

The above line of the code is the true method call which creates the dynamic proxy, the method call takes as parameters a type and an interceptor object. The interceptor object will receive all the methods calls made on virtual methods overridden in the proxy class.

```
internal class MyInterceptor : StandardInterceptor {
     private Simulator manager;
     public MyInterceptor(Simulator manager) {
       this.manager = manager;
     }

     public override object Intercept(IInvocation invocation,

       params object[] args) {
       if (invocation.Method.Name == "Wait") {
          if (args.Length == 0) {
            manager.Wait();
          } else if (args.Length == 1) {
            if (args[0] is long) {
              long t = (long)args[0];
              manager Wait(t);
            } else {
              manager.Wait(args[0] as Event);
            }
          }
       } else {
          base.Proceed(invocation, args);
       }
       return null;
     }
 }
```

The above code is the interceptor type we use to catch the wait method calls. In the DynamicProxy.Net framework, all the calls made on a generate proxy are delegate to the Intercept method of a user-defined interceptor from management. Our decision to use the DynamicProxy.Net framework was made because it was the simplest and quickest way for us to achieve are proof of concept implementation.

6. BENEFITS OF OUR APPROACH

The design approach we used for the implementation of our modeling and simulation solutions has many subtle but very important benefits that we will present in this section. The benefits that we will presents do not add significant value to the semantic capabilities of the modeling solution or the efficiency of the simulation solution but rather “opens the

door" to news possibilities. These new possibilities are enabled because of the separation of concerns that we have achieved between modeling and simulation aspects.

6.1. Perfect Separation of Concerns

As mentioned earlier, to our knowledge, all current modeling/simulation solutions that are based on a framework-oriented approach do not posses a clear boundary between modeling and simulations aspects in their design and/or implementation.

The modeling framework that we have presented possesses no dependencies on a simulation solution implementation. It depends only on a well defined set of modeling concepts and a particular model of computation. Because of this separation, system models that used the modeling framework, by transitivity, are themselves independent of a simulation solution. These models, by the means of .Net, may have multiple "clean" representations as depicted in the diagram below. The importance of these clean models of representation is explained in [5].

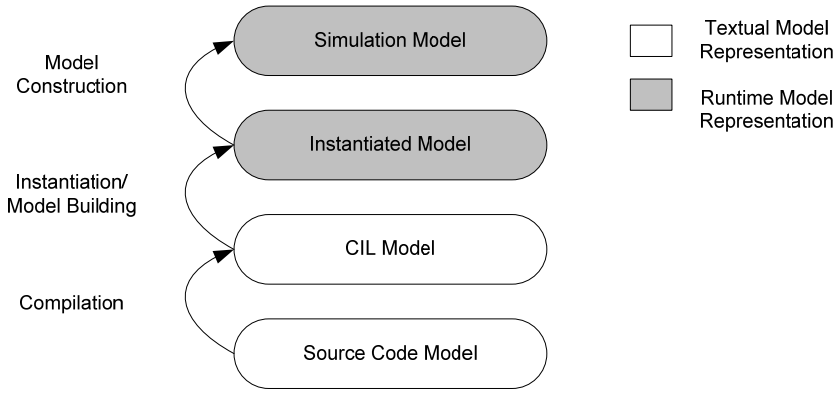


Figure 6. Model representation stacking

We believe that compiled versions of these "clean" models and modules may serve as the backbone for simulation implementation agnostic intellectual property (IP) block-based designs. It is very difficult to achieve true IP block reuse in solutions such as SystemC, for once compiled, IP blocks are statically bound to a specific simulation solution implementation. Separation from specific simulation solutions is fairly important, especially in the context of SystemC, for multiple implementations of the SystemC framework exist and each supports a different toolset. In this context, an IP block which is not independent of a simulation implementation may probably not be used with certain tools. The artificial binding to specific simulation implementations hinders the creation of custom design flows.

6.2. Verification by Interception

The utilization of design patterns such as IoC and Proxy enable a simulation solution to create chains of interceptors which can monitor different aspects of a model under simulation without having to add the verification elements in the model itself or having to create a complex verification enabled layer (API) such as SystemC's SVC.

One can easily imagine an implementation of the **Var** modeling interface we presented which allows a tool external to the simulator to be notified on modification in order to drive linear temporal logic (LTL) expression verification. Another example could be a simulation implementation which allows a verification tool to chain interceptors in a proxy chain in order to monitor the number of method calls on a channel. The combination of a flexible framework design and the reflective capabilities of the .Net framework offers many possibilities for the creation of effective tools at low cost.

6.3. Alternative Model Simulations

Since all models created with the modeling framework are independent of a particular simulation solution implementation, it becomes possible to use a model with different simulators in order to take advantages of alternative implementations. Alternative simulation implementations could offer:

- different performance characteristics

- different monitoring characteristics
- different verification characteristics
- different tool support
- distributed simulation
- software vs hardware simulation for Co-design (Space)
- support heterogeneous Model Of Computation simulation [12, 23].

It is difficult to imagine all the possible simulation solution implementations but what is clear is that a model could be transparently used with various simulation implementations without having to be modified or recompiled, as long as both use the same modeling semantic contract.

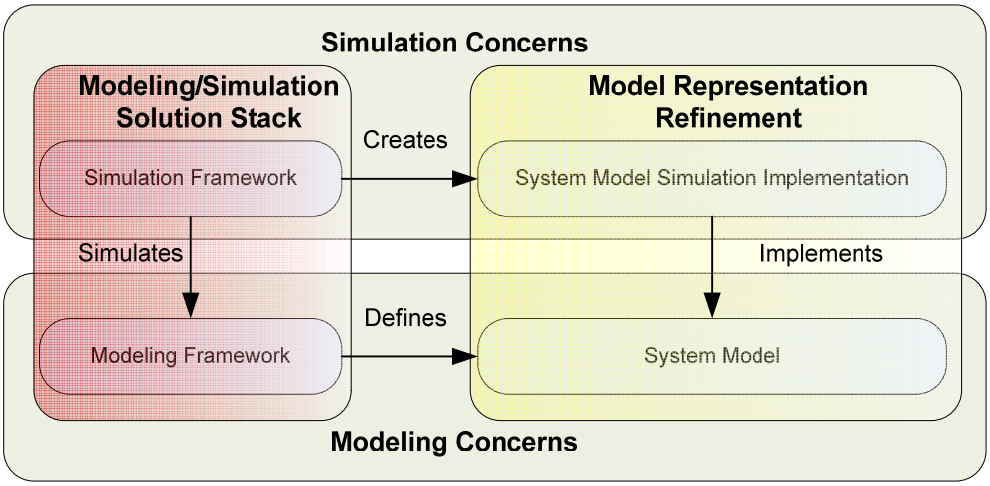


Figure 7. Modeling concerns

Figure 7 illustrates the various relations which exist between the elements of our approach. By using a stack-based approach between the modeling and simulation concerns, it is possible to create various modeling/simulation solutions without affecting models. An emulation solution could be implemented by using the same model analysis techniques based on reflection and IoC but could instantiate the user-defined modules types. The simulation would be achieve not by using a proxy pattern but by orchestrating the execution of multiple hardware elements in the simulation kernel and by updating the variables of the instantiate model, giving the illusion that it is the user model which is being simulated by the kernel. In this context, the instantiate model serves the role of a projection of the emulation.

6.4. Simplicity and Elegancy of Modeling

In our opinion, a model designer should only have to worry about the structural and behavioral aspects of his model and not the implementation constraints of tools which will process the model (*e.g.* simulator). Our statement might seen trivial but in frameworks such as ESys.Net and SystemC, a designer must respectively use the RequestUpdate and Update class methods of the **Channel** base class in order to schedule a primitive channel to be updated during delta cycles in order to perform variable "housekeeping". Should a system designer have to know that the model might be executed with a simulator which implements the concept of a delta cycle? What he really wants is the concept of a variable that can support concurrent reading and writing.

```
class APChannel<t> :PChannel<t>{
    public Event sensitive = new
Event;
    private t currentvalue;
    private t oldvalue;
    private t newvalue;

    public t Value {
      get { return currentvalue; }
      set {
        newvalue = value;
        RequestUpdate(this);
      }
    }

    public override void Update() {
      if (currentvalue != null) {
        if
(!currentvalue.Equals(newvalue)) {
          oldvalue = currentvalue;
          currentvalue = newvalue;
          sensitive.Notify(0);
        }
      } else {
        oldvalue = currentvalue;
        currentvalue = newvalue;
        sensitive.Notify(0);
      }
    }
 }
```

Esys.Net

```
class APChannel <t> : signal<t>{
    private var<t> value;

    public t Value {
      get { return value.Value; }
      set { value.Value = value;}
    }

    public inEvent Sensitive {
      get {
          return value.Sensitive;
        }
    }

  }
```

SoCML

The above side-by-side code comparison demonstrates the simplicity with which a primitive channel may be modeled using our example modeling framework SoCML. The SoCML based model only contains modeling semantics; no simulation related elements are present. As mentioned earlier, the **Var** data type has the semantic meaning of a variable which supports concurrent reading and writing. The implementation of the **Var** concept would probably be similar to implementation approach of the RequestUpdate/Update protocol used in ESys.Net.

Having a simple modeling framework which is free of simulation implementation related information will definitely help designers be more productive. Software tools will also have an easier time analyzing models for they will not have to discard non-modeling related elements. In a perfect dedicated modeling language, each necessary semantic notion would probably be expressed using a small and simple list of key words. We believe that a modeling framework based only on attribute programming, interfaces, and abstract class containing only virtual methods brings modeling framework-based languages much closer to a dedicated modeling language than traditional solutions.

7. FUTURE RESEARCH

The ideas in this article illustrate our vision for the next generation of framework-oriented system modeling languages. This next generation of solutions will be characterized by perfect separation of concerns between modeling and simulation aspects, which may be achieved by applying our design guidelines.

This work opens the door to many other projects such as creating a new SystemC-style modeling solution. It would be interesting to explore the impact of our design guidelines on the design and implementation of a heterogeneous model of computation environment. It would also be interesting to revise the syntax of Metropolis according to the guideline of this work.

Our team is currently working on a redesign of the ESys.Net modeling/simulation framework according to the guidelines we have presented and we are investigating different backend implementation for the simulation engine. We

are also working on the same kind of separation of concerns concept but applied to verification and model constraint aspects.

8. CONCLUSION

Many believe that framework-based modeling/simulation solutions will allow designers to model systems more effectively and will facilitate the creation of custom design flows. Moreover, many believe that framework-based solutions are a good approach to heterogeneous "model of computation" (MOC) and "co-design" simulation. In the mist of all the work which has been done in order to create such framework-based solutions, very few have worked on the design guidelines that such solutions should follow in order to create effective object-oriented frameworks.

In this article, we presented the current lack of separation of concerns which is present in most mainstream modeling/simulation framework-based solutions. We argue about the importance of such separation and its benefits. We present "state of the art" software design patterns and technologies which can promote better separation of concerns between modeling and simulation aspects when applied to modeling/simulation frameworks. Finally we present a novel modeling framework called SoCML which follows our guidelines. By its design, SoCML presents all the benefits which were enabled by the approach:

- simplicity and elegance of the models which use the framework;
- independence of models from a particular simulation implementation allowing their reuse with alternative simulation approaches;
- verification by inception enabled;
- "clean" model representations.

In order to demonstrate the feasibility of our approach, we present a simulation implementation for SocML.

REFERENCES

[1] ITRS 2005. *International Technology Roadmap for Semiconductors, Edition 2003.* Available from: http://public.itrs.net/

[2] David I. Rich, "The Evolution of SystemVerilog," *IEEE Journal Design & Test of Computer*, **20**(2003), p. 82.

[3] THE OPEN SYSTEMC INITIATIVE (OSCI) 2005. *SystemC 2.1 Language Reference Manual*. Available from: http://www.systemc.org/

[4] SYNOPSYS, *COCENTRIC® SYSTEM STUDIO*, 2001.

[5] James Lapalme, El Mostapha Aboulhamid, and Gabriela Nicolescu, "A New Efficient EDA Tool Design Methodology", *Journal of ACM Transactions on Embedded Computing Systems (TECS)*, **5**(2006), p. 408

[6] Nicolas Gorse, Michel Metzger, James Lapalme, El Mostapha Aboulhamid, Yvaon Savaria and Gabriela Nicolescu, "Enhancing ESys.Net with a Semi-Formal Verification Layer", *in Proceedings of the 16th IEEE Intl Conference on Microelectronics (ICM'04)*, 2004 , p. 388.

[7] James Lapalme, El Mostapha Aboulhamid, Gabriela Nicolescu, Luc Charest, Jean-Pierre David, Francois Boyer, and Guy Bois, "ESys.NET: A New Solution for Embedded Systems Modeling and Simulation," *in Proceedings of the ACM SIGPLAN/SIGBED 2004 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'04)*, 2004, p. 107.

[8] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Massachusetts : Addison–Wesley, 1994.

[9] Edsger W. Dijkstra, *Selected Writings on Computing: A Personal Perspective*. New York: Springer-Verlag, 1982, p. 66.

[10] MICROSOFT 2005. *Microsoft .NET Framework.* Available from: http://msdn.microsoft.com/netframework/

[11] ECMA/ISO 2006. ECMA (334-335) and ISO/IEC (23270-23271), *C# and Common Language Infrastructure Standards*. Available from: http://www.ecma-international.org/

[12] *Ptolemy Project*. http://ptolemy.eecs.berkeley.edu/

[13] Peter Bellows and Brad Hutchings, "JHDL: AN HDL for Reconfigurable Systems," *in Proceedings of the IEEE Symposium on FPGAs For Custom Computer Machines*, 1998, p. 175.

[14] Marjan Mernik, Jan Heering, and Anthony M. Sloan, "When and How to Develop Domain-Specific Languages", *in ACM Computing Surveys*. New York : ACM, 2005, p. 316.

[15] Daniel D. Gajski, Jianwen Zhu, Rainer Dömer, Andreas Gerstlauer, and Shuqing Zhao, *SpecC: Specification Language and Methodolog*. New York*:,* Springer, 2000, p. 336.

[16] IEEE 2006. IEEE 1076.3 VHDL.

[17] IEEE 2006. IEEE 1800 SystemVerilog.

[18] Jerome Chevalier, Olivier Benny, Mathieu Rondonneau, Guy Bois, El Mostapha Aboulhamid, and Francois-Raymond Boyer, "Space: A Hardware/Software SystemC Modeling Platform Including an RTOS", *in Source Languages for System Specification*. Massachusetts: Kluwer Academic Publishers, 2004, p. 91.

[19] Griffin Caprio, "Dependency Injection", *MSDN Magazine*, **20**(2005).

[20] Frederic Doucet, Sandeep Shukla, and Rejest Gupta, "Introspection in System-Level Language Frameworks: Meta-Level vs. Integrated.", *in Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, 2003, p. 382.

[21] *The Castle Project - DynamicProxy.Net*. http://www.castleproject.org/

[22] Gabriela Nicolescu, Sungjoo Yoo, Aimen Bouchhima, and Ahmed A. Jerraya, "Validation in a Component-Based Design Flow for Multicore SoCs.", in *Proceedings of the 15th international Symposium on System Synthesis (ISSS)*, 2002, p. 162.

[23] Ahmed Jerraya and Rolf Ernst, "Multi-Language System Design.", in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, 1999, p. 696.