El Mostapha Aboulhamid - Université de Montréal - Canada
Frédéric Rousseau - Laboratoire TIMA UJF/INPG/CNRS - France

System Level Design with .NET Technology

CRC PRESS
Boca Raton London New York Washington, D.C.

Dedication

For Karine, my parents, and all my family, for their help and support - Frédéric Rousseau

To my spouse and my mother, to all those who helped me, influenced me, or endured me throughout all these years, I express my profound gratitude. El Mostapha Aboulhamid

About the editors

El Mostapha Aboulhamid
Université de Montréal - Canada
El Mostapha Aboulhamid is active in modeling, synthesis and verification in hardware/software systems. He obtained an Engineering degree from ENSIMAG, France in 1974 and a Ph.D. from Montréal University in 1984. He is currently professor at Université de Montréal. He worked in the 1980s and early 1990s on built-in-self test techniques, design for testability, multiple fault automatic test generation and complexity of test. He was involved in the current methodology of design of hardware/-software systems since it early beginning in the 1980s and 90s with the introduction of VHDL. He helped to the acceptance of this methodology in Canada, by the collaboration with industrial partners and by delivering intensive courses on modeling and synthesis both in academia and industrial settings. He also collaborated to the standardization of SystemC. He was the director of GRIAO, a multi-university research center which led to the creation of the current ReSMiQ Research Centre. He supervised more than 80 graduate students. He has been General or Technical Program Chair of many conferences: such as ISSS/CODES, ICECS, NEWCAS, ICM, AICCSA. He also served on Steering or Program Committees of different International Conferences. In 2003 his team developed ESys.NET as an environment for modeling and simulation. His is looking into ways of using distributed simulation to overcome this bottleneck caused by simulation of large digital systems. He is also interested in advance software approaches in system level design and reuse. He has multiple collaborations nationally and abroad on different aspects of System On Chip modeling and verification. He has been an invited professor both at Université de Lille and Université de Grenoble in France.

Frédéric Rousseau
Laboratoire TIMA UJF/INPG/CNRS - France
Frédéric Rousseau has been professor since 2007 (and associate professor since 1999) at University Joseph Fourier (UJF) where he teaches computer science and he has been researcher in TIMA lab since 1999. He received the Engineer degree in computer science and electrical engineering from University of Grenoble in 1991 and a Ph.D. in computer science in 1997 from University of Evry (near Paris). His research interest have concerned Systems on Chip design and architecture, and more precisely the design and validation of hardware/software interfaces. He is now focusing on prototyping, software code generation for Multiprocessor System-on-Chip and communication on such systems. He also served on program committees of different international conferences, workshops or symposiums. In 2006, he spent one year of sabbatical at Université de Montréal, working on ESys.NET.

Contributor Biographies

Amine Anane
Université de Montréal - Canada
Amine Anane is a Ph.D. student at the department of computer science and operations research of the Université de Montréal. He received the computer science engineer degree from the Faculty of Science of Tunis in 1998. He has been working as IT consultant for 7 years before joining the Université de Montréal for M.S. degree in computer science. Since he obtained an accelerated admission to Ph.D. program in 2006, he has been with the LASSO laboratory which is interested in the formal design and verification methods of microelectronics systems. His researches are related to the study of a design methodology suitable to formal verification and correct-by-construction incremental refinement.

Guy Bois
École Polythechnique de Montréal - Canada
Guy Bois is a professor in the Department of Computer Engineering at Ecole Polytechnique de Montréal. His research interests include hardware/software codesign and coverification for embedded systems. Guy Bois has a Ph.D. in computer science from the Université de Montréal.

Luc Charest
Université de Montréal - Canada
With a strong C++/software engineering background, Luc Charest, was one of the first to introduce Design Patterns to the System Design domain. Graduated from Université de Montréal in 2004, he then made a postdoc at the LIFL of Université des Sciences et Technologies de Lille (France) in 2005. Some of his current work and other theme interests are operational research and functional programming.

Mathieu Dubois
Université de Montréal - Canada
Mathieu Dubois is a Ph.D. candidate in computer science at the University of Montréal. He holds a B.Ing. and a M.Sc.A. in electrical engineering from respectively the Ecole de technologie supérieure de Montréal and the Ecole polytechnique de Montréal. His research interests include heterogeneous compilation and the acceleration of discrete-event simulations.

Patrice Gerin
Laboratoire TIMA INPG/UJF/CNRS - France
Patrice Gerin received the M.S. degree in Microelectronics from University Joseph

Fourier and is currently working toward the Ph.D. degree from INPG, Grenoble, France. From 1999 to 2005, he has been working as embedded software engineer in the industry. He is currently with the System-Level Synthesis Group in TIMA Laboratory, Grenoble, France. His research interests include Hardware/Software simulation and embedded software validation in MPSoC design.

Nicolas Gorse
Université de Montréal - Canada
Nicolas Gorse obtained a Ph.D. in Computer Science from University of Montréal in 2006. He is currently working within the Analog Mixed Signal Group at Synopsys, Inc. His research interests are simulation, verification and formal methods.

Yann-Gaël Guéhéneuc
Ecole Polythechnique de Montréal - Canada
Yann-Gaël Guéhéneuc is associate professor at the Department of computing and software engineering of Ecole Polytechnique de Montréal where he leads the Ptidej team on evaluating and enhancing the quality of object-oriented programs by promoting the use of patterns, at the language-, design-, or architectural-levels.

James Lapalme
Université de Montréal - Canada
James Lapalme is a PhD candidate at l'Université de Montréal. He has spent most of his graduate research on the application of modern software engineering technologies to embedded systems design. He developed ESys.NET in the context of his master's thesis. Over the past couple of years he has become increasingly interested in the use of Semantic Web technologies for the development of CAD tools. In addition to his academic career, James is a professional in the private sector specializing in the areas of enterprise architecture and enterprise information management.

Michel Metzger
Université de Montréal - Canada
Michel Metzger is a research-and-development engineer at STMicroelectronics Canada. His research interests include system-level design and verification of embedded platforms. Michel Metzger has an engineering degree in Computer Science from Ecole Supérieure en Sciences Informatiques, Sophia-Antipolis, France and a Master of Science from University of Montréal.

Gabriela Nicolescu
École Polythechnique de Montréal - Canada
Gabriela Nicolescu is currently Associate Professor at Ecole Polytechnique de Montréal teaching embedded systems design and real-time systems. She received a degree in Engineering and Ms.S. from Polytechnic University of Romania in 1998. She received her Engineer Doctor degree from the National Polytechnique Institute, Grenoble, France in 2002. Her research work is in the field of specification and validation

of heterogeneous systems, and multiprocessor system-on-chip design.

Frédéric Pétrot
Laboratoire TIMA INPG/UJF/CNRS - France
Frédéric Pétrot is Professor in Computer Architecture at the ENSIMAG, a school of higher education of the Institut Polytechnique de Grenoble since 2004. He is also heading the System Level Synthesis Group of the TIMA laboratory, in Grenoble, France. Prior to this position, Frédéric Pétrot was Assistant Professor in the SoC design Lab of the University Pierre et Marie Curie, Paris, France, where he was a major contributor of the Alliance VLSI CAD System (PhD in 1994 on this topic), and of the Disydent Digital Design Environment.

Yvon Savaria
École Polythechnique de Montréal - Canada
Professor Yvon Savaria holds a Canada Research Chair in architecture and design of advanced microelectronic systems. He has 27 years of experience with IC design and test. He conducted research on design methods of digital, analog and mixed signal integrated circuits and systems. He has extensively published on a wide range of microelectronic circuits and system design methods. He has active collaborative projects with several organizations and was a founder of LTRIM, a Polytechnique spin-off that commercialized an invention resulting from his research at Polytechnique and for which he obtained an NSERC Synergy award in 2006.

Yousra Tagmouty
Université de Montréal - Canada
Yousra Tagmouty received M.Sc. in Computer science from Université de Montréal in 2008. Her thesis describes a meta-model to specify the behaviour of the solutions of design patterns and-or of object-oriented programs. She applied this meta-model to specify several design patterns and to automatically generate the corresponding source code in ESys.NET. She actually works in a software company in Montreal.

Alena Tsikhanovich
Université de Montréal - Canada
Alena Tsikhanovich is the P.D. candidate of the department of computer science and operations research of Université de Montréal. She has received the bachelor degree in mathematics in Belarus State University and the master degree in computer science in Université de Montréal. Her current research projects are related to the domains of hardware/software system modeling and timing verification based on temporal constraint analysis.

Julie Vachon
Université de Montréal - Canada
Julie Vachon is an associate professor of computer science at Université de Montréal. She is a member of the GEODES software engineering laboratory. Her current re-

search interests include formal software specification and verification (model-checking and theorem proving), aspect-orientation, feature interaction detection, transaction models and distributed systems verification.

Contents

Preface

The introduction of VHDL in 1987 and SystemC in 1999 gave a big boost to the Electronic Design Community and played an important role in the development of System Level Design. We were involved with both processes early on. Rich with the experience with these two environments, we wanted to explore new frontiers that can enforce these systems and hopefully constitute a synergy with them. This results in the development of ESys.NET in 2003.

This book had its origin in the overall work done at the Université de Montréal, on the system level design environment named ESys.NET. It is based on the .NET framework and brings a better management of metadata, introspection, and interoperability between tools. The interoperability is one of the most important aspects of frameworks such as .NET. It enabled us to develop for example assertions based observers of ESys.NET models without any interference with the modeler. This can be seen as enabling separation of concerns.

Encouraged by our experience with ESys.NET, we continued our efforts to try to build a bridge between advances in the software community and the needs in the EDA community for new ideas and algorithms. We pursued the development of our environment by exploring new mechanisms such as transaction modeling to help in distributed simulation, or Web Semantics to help with IP (Intellectual Property) reuse.

The collaboration between the SLS group of TIMA in Grenoble (France) and the LASSO group in Université de Montréal was a determining factor in the completion of this work. While the two groups have the same global objectives, they have complementary strengths. The LASSO groups is more focused on modeling and verification, while the SLS group has a valuable expertise in architecture, System on Chip and code generation. Both have also a common interest in accurate and efficient simulation. Sabbatical stays and exchanges helped to strengthen this collaboration.

This work summarizes our efforts and covers three main parts: (a) modeling and simulation, including requirements specification, IP reuse, and applications of design patterns to Hardware/Software systems; (b) simulation and validation, covering Transaction-based models, accurate simulation at cycle and transaction levels, cosimulation and acceleration techniques, and timing specification and validation; (c) practical use of the ESys.NET environment concludes this work.

We would like to thank all the authors for their timely response and the numerous iterations to complete their respective chapters.

Readers are encouraged to visit the companion website http://www.esys-net.org/ and send us their comments to enrich it.

4

Translating Design Pattern Concepts to Hardware Concepts

Luc Charest

Université de Montréal - Canada

Yann-Gaël Guéhéneuc

Université de Montréal - Canada

Yousra Tagmouti

Université de Montréal - Canada

Abstract For half a century, hardware systems have become increasingly complex and pervasive. They are not only found in satellite navigation systems or automated factory machinery but also in everyday cell-phone, parc-o-meter, and car control-and-command systems. This increase in the use of hardware systems led to a revolution in their design and implementation: the chips are becoming more and more powerful, their logics is implemented as software systems executed by the chips, thus helping system designers to cope with their complexity.

These *mixed hardware–software systems* raise the level of generality of the "hardware part" and the level of abstraction of the "software part" of the systems. Thus, they suggest that mainstream software engineering techniques and good practices, such as design patterns, could be used by system designers to design and implement their mixed hardware–software systems.

This chapter presents a proof of concept on "translating" the solutions of design patterns into hardware concepts to alleviate the system designers' work and, thus, to accelerate the design of mixed hardware–software systems. This chapter opens the path towards a new kind of hardware synthesis.

4.1 Introduction

For half a century, hardware systems have become increasingly complex and pervasive. They are not only found in satellite navigation systems or automated factory machinery but also in everyday cell-phone, parc-o-meter, and car control-and-command systems. This increase in the use of hardware systems led to a (r)evolution in their design and implementation: the chips are becoming more and more powerful, their logics is implemented as software systems executed by the chips, thus helping system designers to cope with their complexity.

These *mixed hardware–software systems* raise the level of generality of the "hardware part" and the level of abstraction of the "software part" of the systems. Thus, they suggest that mainstream software engineering techniques and good practices, such as design patterns, could be used by system designers to design and implement their mixed hardware–software systems.

As a variable may match a register, we propose a mapping between design patterns and a hardware implementation. System designers could use this mapping when designing and implementing their mixed hardware–software systems to translate the solution of a design pattern into its appropriate hardware counter-part. Thus, designers would benefit for their systems of the good practices embodied by design patterns from software design.

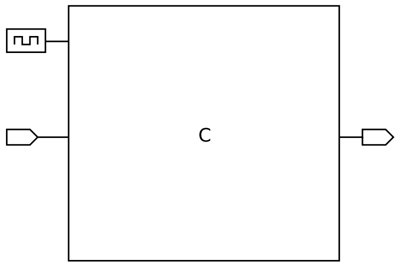
This chapter presents a mapping to "translate" some design patterns into hardware concepts to alleviate the system designers' work and, thus, to accelerate the design and quality of mixed hardware–software systems. It focuses on interesting and challenging concepts to foster future research, without trying to be exhaustive.

Design patterns are "good" solutions to recurring design problems in software design. We only consider the design patterns originally defined by Gamma et al. [89], because these patterns are well-defined, well-known, and the subject of many work in the software engineering community. With the beginning of the 21^{st} century, design patterns began to emerge in the system design domain [51, 23].

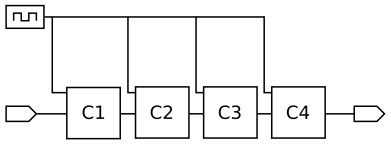
However, the main challenge of mapping design pattern into hardware systems is that existing design patterns relate to object-oriented systems. Therefore, as a first approach of translating design patterns from software into hardware, we consider that we first build an "object system"—a representation of an object into the hardware world. As there are several ways of designing and implementing a processor, there is not only one way to build an object system.

This chapter is not a catalog of *recurring hardware design patterns* because such a catalog should be compiled by the community based on the hardware designers' experience and is therefore out of the focus of this chapter.

Motivating Example. Consider as example the problem of a circuit C that performs a computation, as shown in Figure 4.1a. The classic solution to accelerate the given circuit, at the cost of augmenting its latency, is to *pipeline* this circuit into the sub-circuits $\{C_1, C_2, C_3 \dots C_n\}$, each executing a small amount of the computation in



(a) The problem: a complex circuit taking too much time to execute.



(b) The solution: a pipeline which simplifies the task intosmaller sub-circuits while allowing an augmentation of the clockspeed at the cost of constant latency.

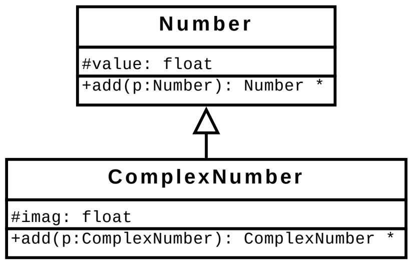
FIGURE 4.1: Motivating example for getting inspiration from software engineering.

parallel, as shown again in Figure 4.1b. (The clock speed could also be increased, resulting in an overall faster execution.)

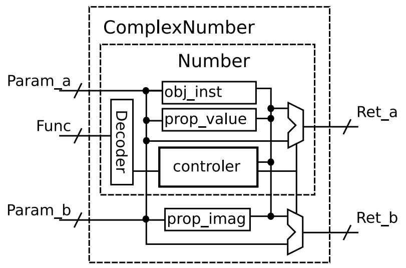
The pipeline architecture has been applied in software engineering to obtain the flexibility to replace particular component seamlessly. For example, it is used to design compilers, where each phase of the compilation corresponds to a component in the pipeline. We believe that other good practices from software engineering could be applied in hardware–software system design and therefore study the feasibility to apply concepts developed in software engineering to the synthesis of hardware systems.

Running Example. In the rest of this chapter, we use the running example of a module `ComplexNumber` to compute and perform operations on complex numbers. Figure 4.2 describe the whole running example, from the software classes to its implementation and instantiation in hardware. We will describe this example step-by-step in the rest of this chapter. In particular, we will use this example to highlight the translation of object-oriented software concepts into hardware concepts, such as inheritance.

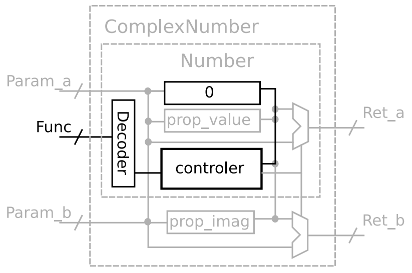
Structure of the Chapter. Design patterns are inherently tied to the object-oriented paradigm. Therefore, in Section 4.2, we present a mapping between software object-oriented concepts and hardware concepts. Then, in Section 4.3, we describe the constraints on our mapping. In Section 4.4, we detail our mapping between design pattern concepts and hardware concepts in the form of a catalog of the most interesting patterns. Such a mapping would be incomplete without a means to translate the patterns into hardware concepts concretely, we therefore present an operational description of design patterns and its use to generate hardware "code" in Section 4.5. In Section 4.6, we describe related work while in Section 4.7, we conclude and introduce future work.



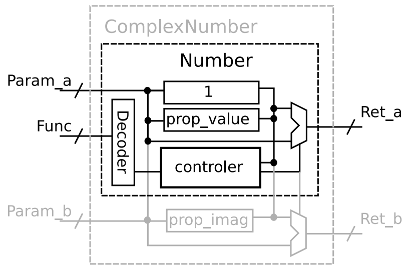
(a) A UML software definition of a `ComplexNumber` class using a `Number` base `class`.



(b) A possible hardware implementation of Object-Oriented inheritance.



(c) The module being in an "not instantiated" state.



(d) The `ComplexNumber` module down-cast into (or instantiated as) a `Number` class.

FIGURE 4.2: Running example of a `ComplexNumber` software class and its hardware module.

4.2 Object-Oriented Translations

The design patterns in Gamma et al.'s catalog [89] are solutions to recurring object-oriented design problems. We therefore present first a mapping between object-oriented concepts and hardware concepts. This mapping will be used in Section 4.4 to map design patterns and hardware concepts.

Essential concepts in object-oriented design and implementation are: objects and their instantiations, methods, inheritance (and casting operations), and polymorphism. We also discuss the cost of generating hardware code from object-oriented code based on our mapping.

4.2.1 Translation of Classes and their Members

In object-oriented programming, the main concepts of interest are `Struct`ures or `Classes`es, which contain fields and related methods. For example, in C++, the distinction between a `struct` and a `class` is the default scope: members of classes are `private` by default while those of structures are `public` by default.

Class members may be of two types: fields, which contain data and define the "state" of a class or of its objects, and method (or constructor) that provide functionalities usually operating on the data contained in the fields. Even if the fields are tied with their methods in the `class`, they are distinct members.

Methods are shared among all the instances of a same `class` while fields are not necessarily. Indeed, fields can be *instance fields*, whose values are unique for a particular object, instance of the class, or *class fields*, whose values are shared by the class and all of its instances.

4.2.2 Translation of Object Encapsulation

Encapsulation is the means by which a class embeds and hides its members. It requires a protection mechanism for the enclosing class to gain the responsibilities over its members and to isolate them from the outside world.

Considering that we are targeting synthesis, with the software being ready to be transformed into hardware, we can treat a base class and its inherited classes similarly to an enclosing class and its encapsulated member. The encapsulated member can be generated directly into the enclosing class or it could be indexed with a pointer to another instance.

4.2.3 Translation of Object Instantiation

An instantiation correspond to the creation of a new object, hence memory allocation for their instance fields. When either a `struct` or `class` are locally declared, their instance fields are allocated on the execution stack. When a dynamic instantiation is requested (with `new` or `malloc`), the instance fields are allocated on the heap.

Unless a class has no fields and therefore provides only methods, its fields must be define in its hardware equivalent. Fields are required to be created in two different situation:

1. At the beginning of the execution of the system, for class fields.

2. At the instantiation of the object, when its constructor gets called.

There are at least four ways of storing instance fields:

As constants, hard-coded as bits. This kind of implementation is interesting for some rare case of static constant fields, which are initialized at the beginning of the execution and which values do not change during execution. Constant fields would be generated with the platform in a ROM (or with a static combinatory circuit) and will have an infinite lifespan.

In registers, distributed in small "modules" that could be generated to corresponds to an object.

In a global RAM memory, probably the best way to store the field values, in a centralized unit.

In a local RAM memory, similar to the previous way, but distributed over the system.

Instantiating class fields, at the beginning of the execution of the system is straightforward because the translator could identify such cases and they can be implemented in hardware as constants or into registers for faster access. The fact that the class fields are shared by all object makes it easy because the fields become unique in the whole system for a given class.

The instantiation of a instance fields is more challenging. The exact moment of a call to a given constructor is not known in advance if not simulated first. For example, nothing forbids the constructor call to be controlled by something like a random draw (e.g.: `Rand()`). Therefore, the time at which the construction takes place, and at which the instance fields must be allocated, cannot be taken for granted and the number of time a given constructor is called could also change between different execution runs.

Therefore, for any realistic system, we cannot pre-compute the exact number and type of each object that will be instantiated in the system. We can only estimate this number and design a system that will be able to contain the most appropriate number of instances of each objects.

4.2.4 Translation of Object Method calls

A method call corresponds to a static function call using an hidden `this` pointers to indicate the object on which to apply the function. For example, the call in Listing 4.1 (Line 13) is the object-oriented equivalent of the function call in Listing 4.2 (Line 12).

The method calls in Figure 4.3a and 4.3b can be translated into a signal sent from one module to another, as shown in Figure 4.3c. Parameters can be sent over the signal data lines and the return values can be sent back as signals as well. If an elaborated structure is to be exchanged, a memory reference, as shown in Figure 4.3d could be sent instead of concrete values.

4.2.5 Translation of Polymorphism

Polymorphism allows behavioral variations based on the class of an object. Polymorphism corresponds to a method defined in several specialized classes with a signature identical to the signature of the method in the base class. When the method of an object is called, depending on the class that was used when the object was instantiated, the method of the appropriate class is called, even if the object was stored beneath a base class reference, as illustrated in Listing 4.3, Line 6 and 13. In this example, `f()` is polymorphic because it is redefined in the inherited class and it is marked as `virtual`. Both `g()` and `h()` are not polymorphic.

```
// Class definition
class List {
  protected:
    class Item *head;
  public:
    // Object-Oriented prototype (method):
    void add(Item &item_to_add);
}

/* Object-Oriented call, the "*this" pointer is "hidden", made
implicit by "my_list", which is the object onto which to apply the
"add" method */
my_list.add(Item(42));
```

Listing 4.1: Object-oriented way; the `Add` method applied on a `List` object.

```
// Structure definition
struct List {
  struct Item *head;
}

// Procedural prototype (function):
void List_add(List *list_to_modify, Item &item_to_add);

/* Procedural call, here the structure onto which apply the behavior
of "add" must be made explicit by passing the reference to the
"List" structure */
List_add_function(&my_list, Item(42));
```

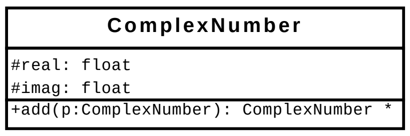
Listing 4.2: Classic procedural way; the call of the `Add` function with the `List` structure passed explicitly.

Polymorphism is usually implemented using a Virtual table (also known as Vtable), which is a table that maps classes with pointers to methods to direct any call to the appropriate method.

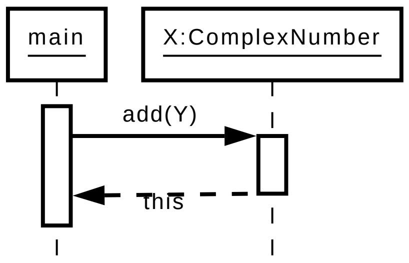
A virtual table is available at compile time, after parsing, before linking, as illustrated in Listing 4.4, which shows the assembly code of the virtual table with names mangling correspondence shown with Table 4.1).

The class of an object is not always known at compile time and the virtual table is persisted in the machine code and preserved for the methods called dynamically. Only virtual methods ends up in the virtual table (Line 106) because ordinary methods can be called directly, they are linked with the object class that is implicit.

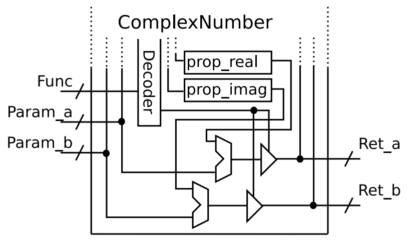
The translation of polymorphism into hardware can be achieved by creating several instance of specialized classes into hardware and then controlling the classes of



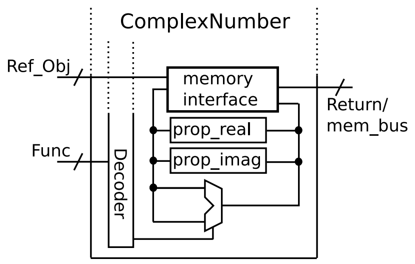
(a) A UML software definition of a `class`.



(b) A software method call.



(c) A possible translation into hardware.



(d) Hardware method call using a memory reference.

FIGURE 4.3: Example of a class representing complex numbers, from its software design to its hardware implementation.

```
class Base
{
public:
  virtual void f(void);
  void g(void);
};

class Derived :public Base
{
public:
  virtual void f(void);
  void h(void);
};
```

Listing 4.3: Polymorphic `f()` method defined in base and derived `class`.

objects by deactivating some part of the module when base classes are needed.

The translation of the Vtable into hardware then becomes straightforward, as exemplified in Table 4.2, because it is similar to a LUT (Look-Up Table). The LUT

```
_ZTV4Base:
        .long   0
        .long   _ZTI4Base
        .long   _ZN4Base1fEv
        .weak   _ZTS7Derived
        .section        .rodata._ZTS7Derived,"aG",@progbits,_ZTS7Derived,comdat
        .type   _ZTS7Derived, @object
        .size   _ZTS7Derived, 9
```

Listing 4.4: The vtable with the assembly language excerpt of the code of Listing 4.3.

TABLE 4.1: Translation of mangled names of Listing 4.3.

Mangled identifier	*Result returned by* `c++filt`
`_ZTV4Base`	vtable for Base
`_ZTI4Base`	typeinfo for Base
`_ZN4Base1fEv`	Base::f()
`_ZTS7Derived`	typeinfo name for Derived

TABLE 4.2: Possible method (and polymorphism) encoding for the `ComplexNumber` module.

Class type	*Method*	*Code assigned*
`Base`	`f()`	0
`Base`	`g()`	1
`Derived`	`f()`	2
`Derived`	`h()`	3

becomes a decoder; the numbers can be attributed sequentially to each newly discovered method upon parsing during code generation.

4.2.6 Translation of Inheritance and Casting Operations

Inheritance and polymorphism help bringing communality and variation [61]. In software engineering, inheritance is achieved by adding methods and properties to the base inherited structure. We reuse this idea in hardware by generating a module for each specialized `class` at the end of the inheritance tree (all leaf classes of the inheritance tree).

Each of these specialized modules contains its parent class, which would in its turn, recursively contain all of its parents, as shown in Figure 4.2b. A special register would then be generated within each class module to hold the type of the module, `obj_inst` as shown in our example. When the module is not instantiated, this number is 0, in Figure 4.2c, indicating all the data in the local registers/memories are invalid.

The proposed mechanism enables also to typecast an object into an inherited class easily by changing the `obj_inst` number to the one indicating the type of the new class. If the class is down-cast, as illustrated in Figure 4.2d, the invalidated registers can still holds some valid values (masked by the downcast) and could be restored when the object is cast back to its original class.

4.3 Constraint and Assumptions for Design Pattern Synthesis

The translation of design patterns from software to hardware is subject to one constraint and three assumptions that we present now.

4.3.1 Constraint: Dynamism of the Hardware

Hardware systems used to be static: sets of wires and components "hardwired" together to perform specific computations. For some times now, hardware systems blend altogether with their software systems to benefit from the dynamic nature of the the software. End products are more customizable with flash memories and configurable with small Web servers implemented in embedded systems as software systems running on a generic hardware core.

Traditionally, objects are generated in memory, having a generic processor performing method calls and fields accesses. Although such a solution is the usual way of reaching the dynamism found in software systems, it is an extreme case that relies on a "pure" software implementation and a generic processor and that is therefore uninteresting in the context of this chapter.

We are interested in implementing design patterns using a "pure" hardware system, which we could define as a hardware system with as less software as possible, and which would potentially bring faster execution at the cost of specializing the hardware. Such solution is more desirable for embedded systems, where the computations or application are unlikely to change.

Naturally, this solution would also work for mixed software–hardware systems (e.g.: FPGAs) which are consequently implicitly be covered by this chapter, because such solution does not directly implement (or emulate) an object-oriented system: we assume a more conservative hardware system, thus guaranteeing that the translation would work on more dynamic hardware systems.

4.3.2 Assumption: Compiled Once

If we were to target an FPGA for our compiled system, it is possible to change the hardware on the fly, thus easing the task of implementing polymorphism.

Such a solution complies with our constraint of obtaining a “pure” hardware solution, even though changing the nature of the hardware on the fly requires more logic gates (as the control logic of the FPGA cell blocks). Indeed, changing the hardware only requires more knowledge at the start of the system.

This knowledge is available if we limit our discussion to such a case where the software system is known and is available as software code, ready to be transformed into hardware by an hypothetical “software to hardware” compiler.

If we are to generate a solution for an ASIC (Application-Specific Integrated Circuit), we have to assume that the translation into hardware will occur once and that the nature of the hardware—unless designed for—can not be changed.

Hence, in this chapter we shall focus the most restrictive platform type, and restrict ourselves to a one-time compilation and synthesis, no dynamic compilation or synthesis.

4.3.3 Assumption: Limited Number of Objects

We could also create several hardware modules to simulate an object-oriented software system, each module matching a `class` and its inheritance tree. We prefer, however, to reuse specialized modules and use them as base modules. In order for each class to be instantiated at least once in our system, we can them assume the number of module must be at least, the number of leaf classes of our system.

Let n be the number of classes in our system at compile time. Assuming that no new classes can be added after generation and that we have a complete binary common-rooted balanced inheritance tree, there are $\frac{n+1}{2}$ leaf classes, the minimum number of modules that must be generated in the hardware.

By generating more hardware modules for a class, we bring parallelism in the system by enabling the existence and computation capability of several objects at the same time. As mentioned earlier, unless the hardware is capable of mutating into another class, once generated, the number of active class in the system at the same time is limited by the number of time the designer instantiate a specific class. This limitation means that a constant must be defined for each class, indicating the maximum number of active objects allowed in the system at a same time.

Computing the maximum of number of active objects is in general impossible, because objects can be created dynamically in software systems and with no other constraints but the size of the memory. For example, a large and unknown number of objects could be created to compute a complex scene in a ray tracer. Yet, it is possible to run the software systems in a set of reasonable scenarios and obtain an insight on the maximum number of objects of its classes. Such practice is already used when developing for example software systems for cell-phones.

4.3.4 Assumption: Pattern Automatic Recognition Problem

Although design patterns are quite formally described with Gamma et al. [89], they were not meant to be defined into a computer language and parsed in an automated way. They express generic solutions to recurring design problems: they cannot be easily automatically identified in a software system.

Since a design pattern can have several variants, identifying occurrences of the pattern in a software system becomes a challenge of its own, which has been tackled by the software engineering community as early as 1998 [206].

Therefore, we assume that we know explicitly which design patterns have been used to implement a software system and which classes play some roles in their occurrences.

4.3.5 Translation Cost versus Performance

An optimization phase can occur to reuse part of the behavioral synthesis process. For example, the controller for the `ComplexNumber` class could use only one ALU, at the cost of a more complex controller module.

In Figure 4.3c, we show a solution to our running example where its behavioral parts, the ALU, are duplicated. Such solution provides more parallelism but at the expense of more hardware.

In Figure 4.3d, we depict an alternate solution where the behavioral part is reused for several distinct method. The need of buses then arises and complexifies the logic circuits of the overall unit (not shown in the figure). Yet, this solution saves on hardware, at the cost of serializing the operations. This solution, in worst case, corresponds to the execution on a classic mono processor architecture. A threshold could be set by the designer generating the hardware system, indicating how many processing modules are to be generated to accelerate the overall architecture.

Another part of reuse could be achieve by replacing local registers by a local memory that could hold an array of objects of the same inherited branch. Let n be the number of distinct classes in a given branch, we can then pose $\{i \mid 1 \leq i \leq n\}$ to be a number identifying each class. Let x_i be the number of living objects required for class type i. The minimal required memory size is then defined by Equation 4.1.

$$\text{memory size} = \sum_{i=1}^{n} x_i \times \text{memory usage of}(i) \tag{4.1}$$

For example, in the running example, the architecture could be configured to hold, in the same `ComplexNumber` module, x_0 objects of the `Number` class and x_1 objects of the `ComplexNumber` class. Such a solution means larger amounts of memory to hold more objects in a same module, at the cost of creating a execution bottleneck if the number of executing units in the module is low. Local buses of units will also be a bottleneck if the number of objects contained by a module is high.

It is advisable to use a mixed approach, where one could generate several modules of the same kind, with each a small memory capable of handling several objects at a same time, to distribute the computations whenever possible while minimizing the hardware cost. The acceleration could be thus maximized, especially with massively parallel systems, where objects are relatively independent.

4.4 Design Pattern Mappings

We now describe some interesting patterns using the mappings of object-oriented concepts into hardware concepts and the constraint and assumptions described before.

4.4.1 Creational Patterns

Creational patterns are the patterns related to the dynamic creations of objects. As explained in Section 4.3.1, we consider hardware as being more static than dynamic. Applying these kinds of dynamic patterns to "pure" hardware is challenging because of the static nature of the hardware. Therefore, we present the mapping of two characteristic creational patterns into hardware: the Prototype and Singleton design patterns.

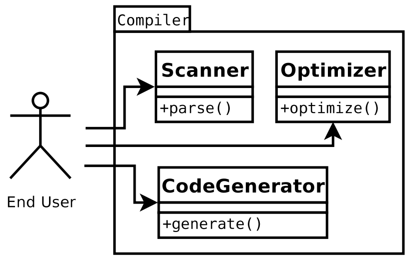
Prototype is a pattern that provides a "typical" instance that can be copied before being customised, with the help of the public method `clone()`. It allows a class to have a default instantiation, and avoid having to call several (maybe complex) methods to initialize an object with its default values.

The prototype pattern corresponds to a ROM (Read Only Memory) that can hold a block of data containing the prototype. Upon call of the `clone()` method, an object is allocated and the data is copied into the newly created instance, creating a new object based on the prototype. The new object will typically have to evolve in time and should be allocated in a RAM or in a register as described in Section 4.2.3

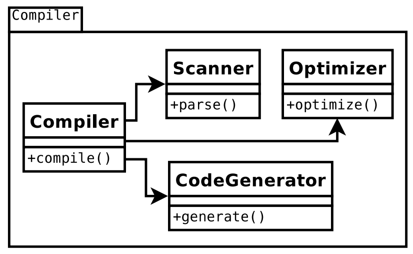
Singleton is a pattern that restricts the number of objects of a class to one. In a software system, where a class can usually be instantiated at will, the need quickly arises to ensure there is only one instance of a certain class that is shared by all other objects of the system, when a second object of the same class could cause miscomputations or crashes (e.g.: a multi-threading controller).

Implementing the Singleton pattern is usually achieved by hiding the constructor from the outside world and providing a class method to obtain the unique object (e.g.: `instance()`, `get_new()`, `get_instance()...`). Any other object is forced to use the class method supplied to get the unique instance of the Singleton class. The class members (and inherited) still have access to the constructor.

In terms of hardware, this pattern is easily implemented by directing the synthesis to generate only one object of a class. A Boolean flag can also be in-



(a) The problem: a complex compiler system.



(b) A Facade class "Compiler" which lighten the use of the system.

FIGURE 4.4: Example of a Façade.

cluded in the controller (for example the one in our inheritance example in Figure 4.2b) to check that the object has been instantiated, and if so, to raise an error with the calling entity to indicate it can not provide a new instance.

4.4.2 Structural Patterns

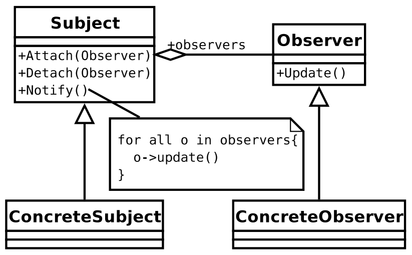
Structural patterns are useful to create the design of a software systems. Beck [29] pointed out that design patterns generate architectures. We describe two typical structural design patterns that are useful to make object interact seamlessly and to isolate a set of objects from the rest of the system.

Adapter is the software equivalent of a wrapper. The goal of the Adapter pattern is to enable the interconnection of several directly incompatible objects by delegating method calls to the appropriate (incompatible) methods either by using multiple inheritance or object composition.

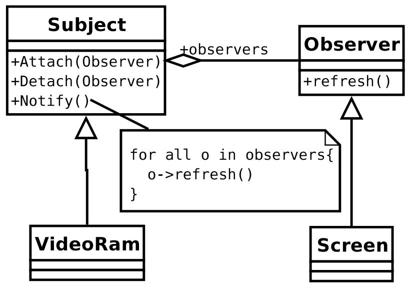
In terms of hardware, it is not rare to see wrappers constructed around IP blocks (Intellectual Properties blocks). As for the software pattern, wrappers may be used to enhance, break into several subcomponents, reroute, or even disable some behavior (or structure) of the component that they are "masquerading" by intercepting part of the communication with the external entity.

Façade is a class that hides the complexity of a whole sub-system into a single object. The classical example of a Façade is a compiler, as shown in Figure 4.4a, where several compilation steps are implemented with several different objects of various classes, each having distinct responsibilities. Façade helps to separate a functionality and to segment the code into simpler parts that are easier to maintain.

The Façade is the class "Compiler" in our example in Figure 4.4b) that is inserted between the system and the external world and acts as an interface to the system. The cost of using a Façade is an extra level of indirection and



(a) Generic layout of the Observer pattern.



(b) The Observer ⇒ ESL Configuration (Memory, Screen, Bus, CPU, DMAs. . .

FIGURE 4.5: Example of an Observer.

the extra burden of updating the Façade when a major change occurs in the system. Moreover, the Façade is sometime blamed for quickly getting big and may lead to an entanglement when it needs to be tightly coupled with a lot of other classes.

In terms of hardware, a Façade corresponds to an interface, where a protocol is defined to access a more complex system. Usually pins are created that might corresponds directly to some internal components, but the interface is usually simplified to reduce its complexity.

A bus can be considered as a Façade as it usually gives access to (while caching access to) a whole complex system. It also usually provides a simple interface (rather than have the external system communicating with every other subsystem.)

4.4.3 Behavioral Patterns

Finally, the last category of patterns include patterns related to the behaviour of objects at runtime.

Observer is a pattern that allows a Subject to notify its Observers when some of its data change thus ensuring consistency among the Observers, as shown in Figure 4.5a.

In terms of hardware, a memory can be considered as a Subject that is observed, as illustrated in Figure 4.5b. The Observers are all the different components that needs to access the memory data (CPU, DMA, peripherals. . .). The bus along with its communication protocol form the “contract” that matches the software interface using inheritance and polymorphism mechanism. We show a screen as the observer in our example.

4.5 Operational Description of Design Patterns

To operationalise our mapping between software design patterns and hardware systems, we choose Esys.NET [138, 139]. Esys.NET is a system design environment (similar to SystemC) based on C♯ and the corresponding .NET framework (rather than C++).

Design motifs are the "Solution" parts of design patterns. They are what developers actually implement in their systems when using design patterns. The generation of design motifs for Esys.NET require a means to describe design motifs in a form that can be manipulated by a computer to perform code synthesis into hardware.

We use the Pattern and Abstract-level Description Language (PADL) as formalism to describe design patterns. We first present PADL. Then, we introduce MIP, an extension to PADL to describe more precisely the behaviour of the methods declared in a motif. Finally, we show the use of PADL and MIP to generate Esys.NET code on the Observer design pattern.

4.5.1 PADL in a Nutshell

PADL is a meta-model that can be used by developers to describe design motifs and object-oriented software systems. A meta-model is essentially a set of classes whose instances represent a model. The methods of the classes in the meta-model describe the semantics of the model. Consequently, PADL provides a set of classes representing constituents of design motifs and the methods required to instantiate and link the instances together in a meaningful way.

Figure 4.6 shows a UML-like class diagram representing the architectural layers of the PADL meta-model, their main packages and classes, and the design patterns used in the design.

The diagram decomposes in three horizontal parts representing three different layers of services: First, `CPL` (Common PADL Library); Then, PADL; Finally, `PADL ClassFile Creator`, `PADL AOL Creator`, POM, and `PADL Analyses`. The first layer, `CPL`, provides utility classes and libraries used across PADL.

The second layer, PADL, provides the meta-model to describe models of systems and motifs. The meta-model defines the interfaces (and implementation classes) of the possible constituents of motifs, for example, `IDesignMotif`, whose instance are motifs and `IClass`, whose instances describe the classes suggested by a motif. These instances are combined to describe motifs and subsets of their behaviours.

The `padl.kernel` and `padl.kernel.impl` packages declares respectively the types of the constituents (as Java interfaces) and their implementations.

The PADL meta-model is at the heart of the Ptidej project (Pattern Trace Identification, Detection, and Enhancement in Java) to evaluate and to enhance the quality of object-oriented software systems, promoting the use of patterns, either at the language-, design-, or architectural-levels. In particular, it has been extensively used to identify occurrences of motifs in systems, for example in [103].

4.5.2 PADL in Details

Figure 4.7 shows the classes and main methods of the constituents of the PADL meta-model. Essentially, the meta-model divides in four parts. The first part includes all the possible constituents (inheriting from `Constituent`) of the structure of a system or a motif. These constituents include different types of entities, `Interface` (interface à la Java) and `Class` (classes found in C++ or Java); methods and fields; parameters.

The second part includes add constituents to refine a model of a system or of a motif with a comprehensive set of binary class relationships. These relationships are important because the interaction among classes and their objects in design motifs are often described in terms of such relationships. The relationships include, from less constraining to the more constraining, the `Use`, `Association`, `Aggregation`, and `Composition` relationships [102]. The `Creation` relationship is also available to describe that objects of a class instantiates objects of another class.

The third part includes the constituents specific to the descriptions of design motifs. A design motif `DesignMotif` is described in terms of its participating classes `Participants` which could be played by classes (`ClassParticipant`) or interfaces (`InterfaceParticipant`). Any participant can declare elements as defined in the part one and two of the meta-model.

Finally, the fourth part includes the constituents specific to the description of a `ProgramModel` and its possible set of `MicroArchitecture`s that are the concrete manifestations of a `DesignMotif`. A micro-architecture knows which of its consistent plays which role in a `DesignMotif`.

We use the Abstract Factory design pattern to manage the concrete instantiation of the constituents of PADL. The concrete factory, class `Factory`, implements the Singleton design pattern. We use the Builder design pattern to let the parsers choose the constituents to instantiate, through the `Builder` class. We use the Visitor design pattern to offer a standard mean to iterate over a model or a subset of a model, the `padl.visitor` package provides default visitors. The `padl.pattern` and `padl.pattern.repository` packages define several prototypal models of well-known design motifs, which we can clone and parameterise, using the Prototype design pattern.

The third layer contains several separate projects:

- Parsers for Java class-files and `AOL` files (`PADL Java` and `AOL Creator`). These parsers are independent of the meta-model and new parsers for other programming languages can be added seamlessly using the Builder design pattern.

- A metric computation framework (POM), in which we use the Singleton design pattern. POM decomposes in a set of primitives defined in terms of the meta-model constituents. These primitives are combined using set operators to define metrics.

- A repository of analyses based on the meta-model, in which we use a simpler version of the Command design pattern. An analyse is invoked on a model of

```
public class Observer extends BehaviouralMotifModel implements
        PropertyChangeListener , Cloneable {

    private IClass subject , concreteSubject ;
    private IInterface observer ;
    private IDelegatingMethod notify ;
    private IMethod update , getState ;

    public Observer() throws CloneNotSupportedException ,
            ModelDeclarationException {

        super("Observer");
        this.setFactory(Factory.getInstance());

        // Interface Observer
        this.observer = this.getFactory().createInterface("Observer");
        this.update = this.getFactory().createMethod("Update");
        this.observer.addConstituent(this.update);
        this.observer.setPurpose(MultilingualManager.getString(
            "Observer_PURPOSE",
            Observer.class));
        this.addConstituent(this.observer);
```

Listing 4.5: The Observer design motif using the PADL meta-model: declaration of the Observer role.

a software system or of a pattern and returns a (potentially modified) model when the analysis is done. Reflection is used by the repository to build the list of available analyses dynamically.

4.5.3 PADL by Examples

PADL has been used to develop a library of design motifs from the 23 design patterns by Gamma et al. [89], including Chain of Responsibility, Composite, Observer, Visitor... For example, we show with the code of Listing 4.5 the Observer design motif using the PADL meta-model. The following PADL code systematically instantiates constituents of the meta-model according to the motif as suggested by Gamma et al., see Figure 4.8.

We show in Listing 4.5 the declaration of the Observer design motif, as a class `Observer`. The motif declares an interface `Observer` that plays the role of Observer in the motif. The interface is built using a `Factory`.

In Listing 4.6 we show the declaration of the Subject role as a `Subject` as a class. This class is abstract and is associated, using an embedded aggregation `ContainerAggregation`, to the previously declared `Observer` class. The `Subject` class also declares a `Notify` methods that delegates its call, through the aggregation, to all the subject's observers.

Listing 4.7 illustrates the declaration of the role of Concrete Subject as a class `ConcreteSubject` that declares a method `getState`. The concrete subjects inherits from the subject and assumes all its interface.

Finally, Listing 4.8 shows the declaration of the role Concrete Observer as a class

```
        // Association observers
        final IContainerAggregation anAssoc =
            this.getFactory().createContainerAggregationRelationship(
                "observers",
                this.observer,
                Constants.CARDINALITY_MANY);

        // Classe Subject
        this.subject = this.getFactory().createClass("Subject");
        this.subject.setAbstract(true);
        this.subject.addConstituent(anAssoc);
        this.notify =
            this.getFactory().createDelegatingMethod(
                "Notify",
                anAssoc,
                this.update);
        this.subject.addConstituent(this.notify);
        this.subject.assumeAllInterfaces();
        this.subject.setPurpose(MultilingualManager.getString(
            "Subject_PURPOSE",
            Observer.class));
        this.addConstituent(this.subject);
```

Listing 4.6: The Observer design motif using the PADL meta-model: declaration of the Subject role.

```
        // Classe Concrete Subject
        this.getState = this.getFactory().createMethod("getState");
        this.concreteSubject = this.getFactory().createClass("ConcreteSubject");
        this.concreteSubject.addInheritedEntity(this.subject);
        this.concreteSubject.setPurpose(MultilingualManager.getString(
            "ConcreteSubject_CLASS_PURPOSE",
            Observer.class));
        this.concreteSubject.addConstituent(this.getState);
        this.concreteSubject.assumeAllInterfaces();
        this.addConstituent(this.concreteSubject);
```

Listing 4.7: The Observer design motif using the PADL meta-model: declaration of the Concrete Subject role.

`ConcreteObserver`. This class is associated to the concrete subjects through another aggregation. It declares an `update` method that is being called by the concrete subject `notify` method when appropriate and that fetches the concrete subject's changes through a call to its `getState` method.

An instance of the `Observer` class is *an instance* of the Observer design motif, which can then be parameterised to fit a given implementation. This parameterised instance can be used to identify occurrences of the motif in a system or to generate source code.

4.5.4 MIP

The PADL meta-model has been extended with additional constituents to describe the inner working of the methods of systems and motifs. This extension to the meta-

```
        final IContainerAggregation a2Assoc =
            this.getFactory().createContainerAggregationRelationship(
                "subject",
                this.concreteSubject,
                Constants.CARDINALITY_ONE);

        // Classe Concrete Observer
        this.notify =
            this.getFactory().createDelegatingMethod(
                "Update",
                a2Assoc,
                this.getState);
        this.notify.setComment(MultilingualManager.getString(
            "DELEG_METHOD_COMMENT",
            Observer.class));
        this.notify.attachTo(this.update);
        this.concreteSubject = this.getFactory().createClass("ConcreteObserver");
        this.concreteSubject.setPurpose(MultilingualManager.getString(
            "ConcreteObserver_CLASS_PURPOSE",
            Observer.class));
        this.concreteSubject.addImplementedEntity(this.observer);
        this.concreteSubject.addConstituent(a2Assoc);
        this.concreteSubject.addConstituent(this.notify);
        this.concreteSubject.assumeAllInterfaces();
        this.addConstituent(this.concreteSubject);
    }
}
```

Listing 4.8: The Observer design motif using the PADL meta-model: declaration of the Concrete Observer role.

model, called MIP, is necessary to describe the behaviour of design motifs more precisely than with PADL alone.

MIP proposes new constituents implementing the interface `IConstituent-OfMethods` to describe the various statements that can be used to define the behaviour of methods. This set includes: `IMethodInvocation`, `IParameter`, `IConditional`, `IInstantiation`, `IAssignment`. Figure 4.9 shows the extension of the PADL meta-model with MIP.

Essentially, the PADL meta-model was refactored to distinguish constituents of methods using the interface `IConstituentOfMethods`. The MIP extension provide a set of such constituents of methods. This set is sufficient to describe several behavioural and creational design motifs more precisely than with PADL alone.

For example, using PADL extended with MIP, the description of the Observer design motif would be extended with the code shown of Listing 4.9 code:

This code describes in more details the behaviour of the `notify` method. Thus, with MIP, it is possible to describe completely the structure and the behaviour of behavioural, creational, and structural design motifs.

4.5.5 ESys.NET Code Generation

The PADL meta-model provides an implementation of the `Visitor` design pattern that allow any client to write visitor to traverse the constituents of a model. We

```
IBlock block = StatementFactory.getStateInstance().createBlock();
this.notify.addConstituent(block);
IIterator iterative =
    StatementFactory.getStateInstance().createIteratorS(this.update);
block.addConstituent(iterative);
IMethodInvocation invocation =
    Factory.getInstance().createMethodInvocation(2, 1, 1, this.subject);
invoc.addCallingField(this.observer);
invoc.setCalledMethod(this.update);
iterative.addConstituent(invocation);
```

Listing 4.9: Extending the the description of the design motif Observer using PADL extended with MIP.

implement such a visitor to generate Esys.NET code from the extended models of design motifs.

4.6 Related Work & Background

4.6.1 Object Oriented Synthesis & Patterns in Hardware

The synthesis of complex C structures has been discussed by [181] and they claim at the end of the article that their methodology can be applied for more complex C++ structures.

Some hardware designs for Object Oriented paradigm have been put forward, especially an Object Oriented processor by [124]. They discuss on an interesting hardware object allocation strategy, although their approach analysis was limited to a global shared memory.

Some patterns were used for hardware modeling as in [64].

4.6.2 Original Patterns

Original Design Patterns were introduced by [89]. Design Patterns express structured and elegant solution (based on the experience of software engineers) applied to object oriented commonly encountered problem.

Design Patterns are sometimes critiqued for a lack of coherency in their interrelations, and blamed for degradation of performance by rising overall design complexity. Despite these disputed drawbacks, they bring other interesting benefits such as:

- clarification of object responsibilities,
- reduced class couplings,
- enhance code genericity,

- augment reusability of classes and algorithms...

Patterns are classified under three major groups:

Creational Patterns are solving problems related to class instantiations. Usually, each given objects know how to instantiate itself. With these patterns, the instantiation responsibility is often delegated to other classes. The creation of complex objects is more structured and more flexible.

Structural Patterns are solving problems related to class structures and interrelations. They help creating more dynamic and flexible class constructions.

Behavioral Patterns are solving problems related to class functionality. Usually, a class contains the implementation of the functionality of each of its instances. Behavioral patterns helps to isolate object comportment from the class definition, bringing a more flexible approach.

4.7 Conclusion

We discussed relations between and matches between some of the Design Pattern in a software form, and their various correspondence in hardware. With the help of such thing as the *Pipeline* pattern, we showed that Design Patterns are not only software specific, but are already present in the domain and should be better outlined.

We presented a specialized object system which can be implemented in "pure" hardware in order to reproduce the behavior of a generic object system running on a processor. We also discussed on how every Object-Oriented aspects can be integrated into hardware, using our object system as examples.

We introduced Esys.NET, a new System Design platform based on C$\sharp$, along with PADL, a Design Pattern framework into which Patterns can be defined and used in order to generate code.

Future area of interests is to further develop the object-oriented system in order to implement a full scale prototype on an FPGA.

The system design community needs to gather the experience they collectively possess into a hardware focused pattern catalog in order to stop reinventing the wheel, and drive the reuse of well known and proofed solutions. This chapter is a first step into the right direction, but only with the help of a thriving community, will we succeed in building a strong collaborative tool based on Design Patterns.

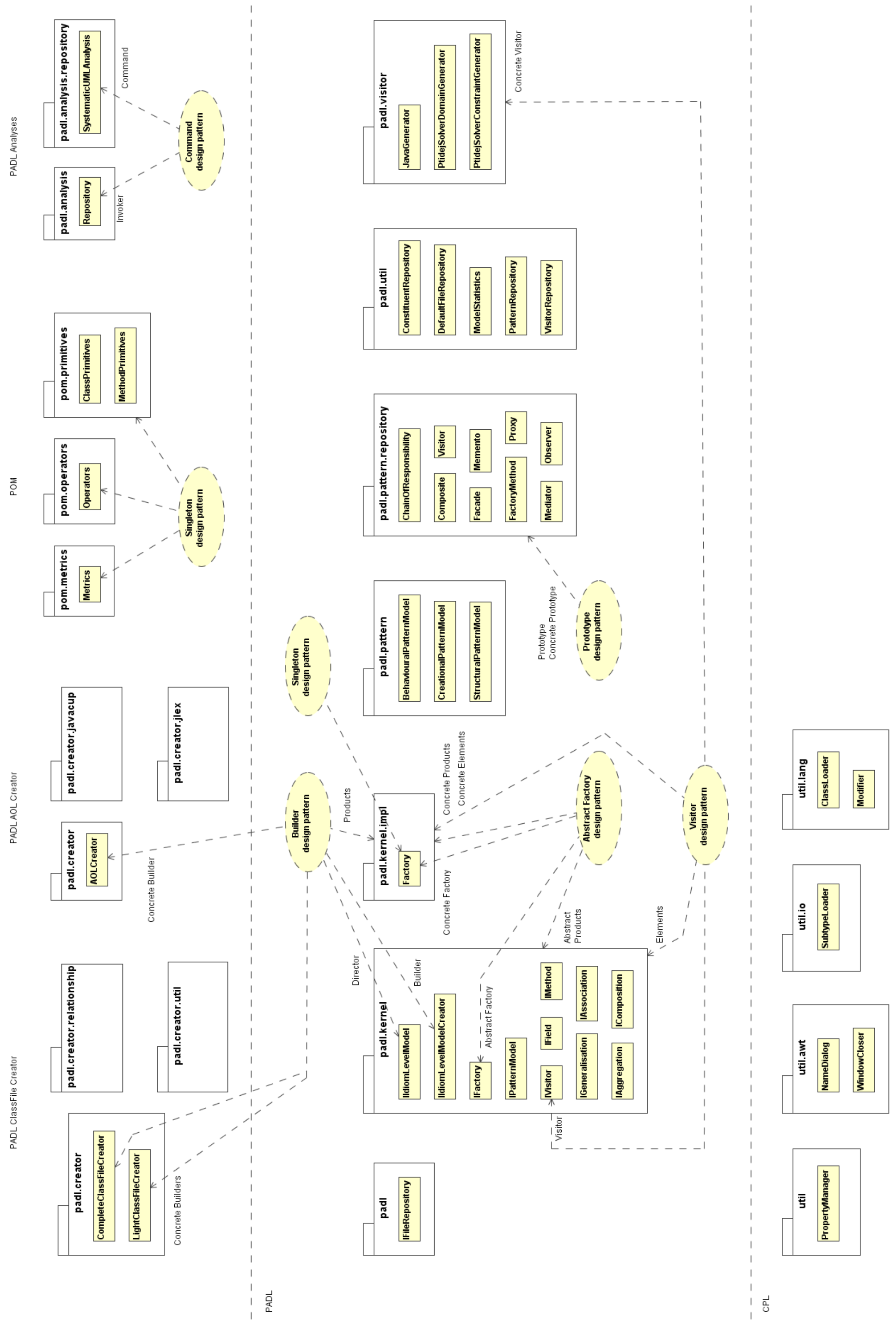


FIGURE 4.6: The PADL meta-model layers.

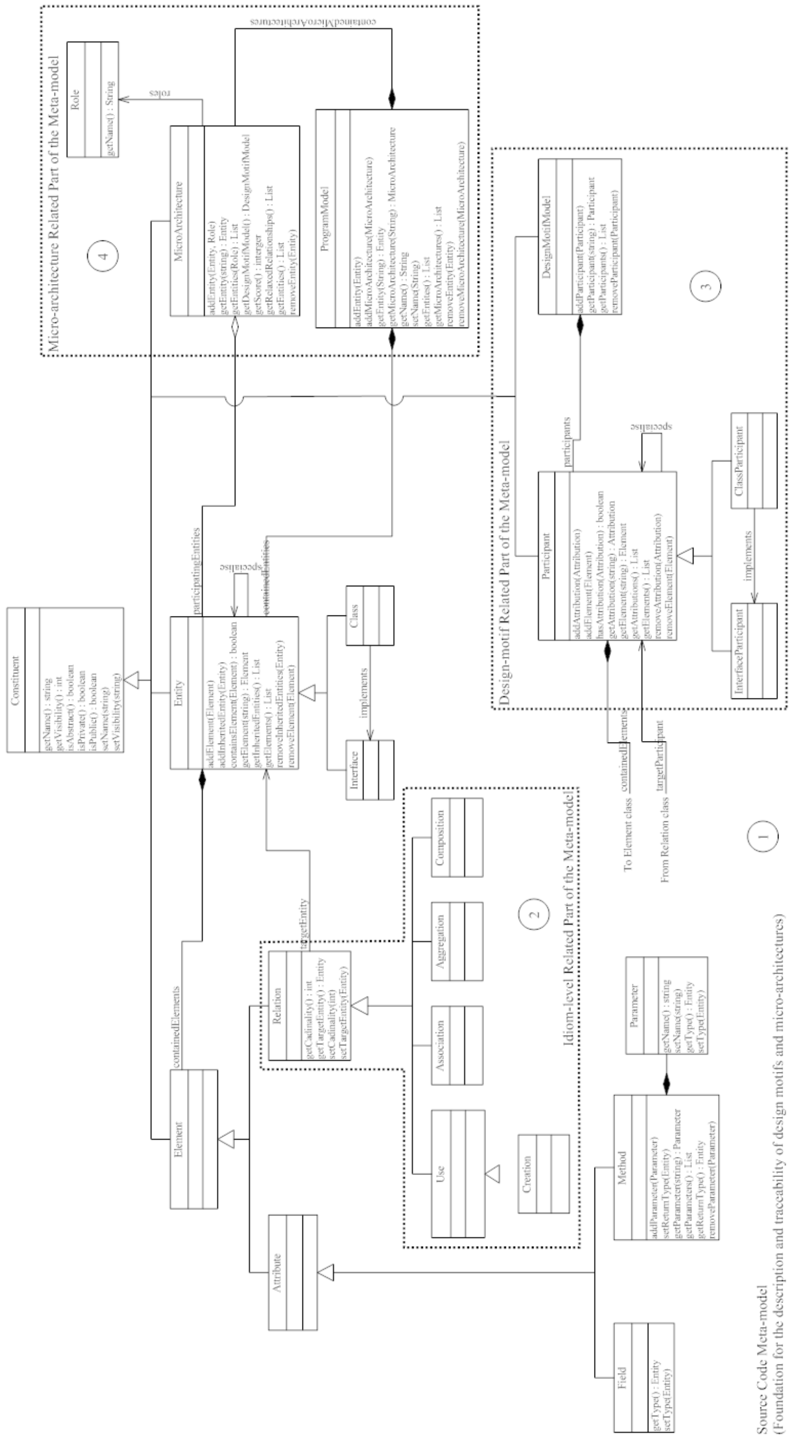


FIGURE 4.7: The PADL meta-model.

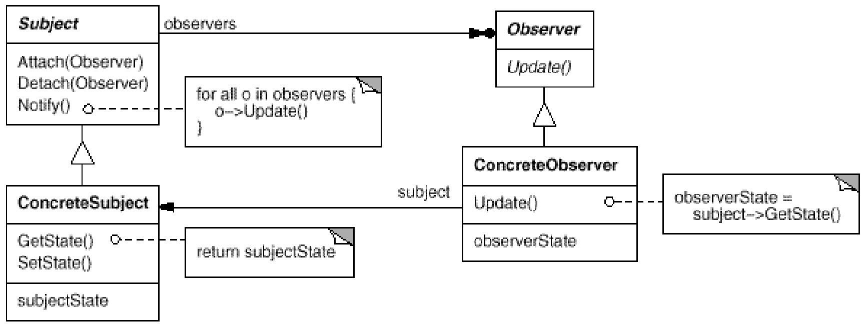


FIGURE 4.8: The Observer design motif (from [89]).

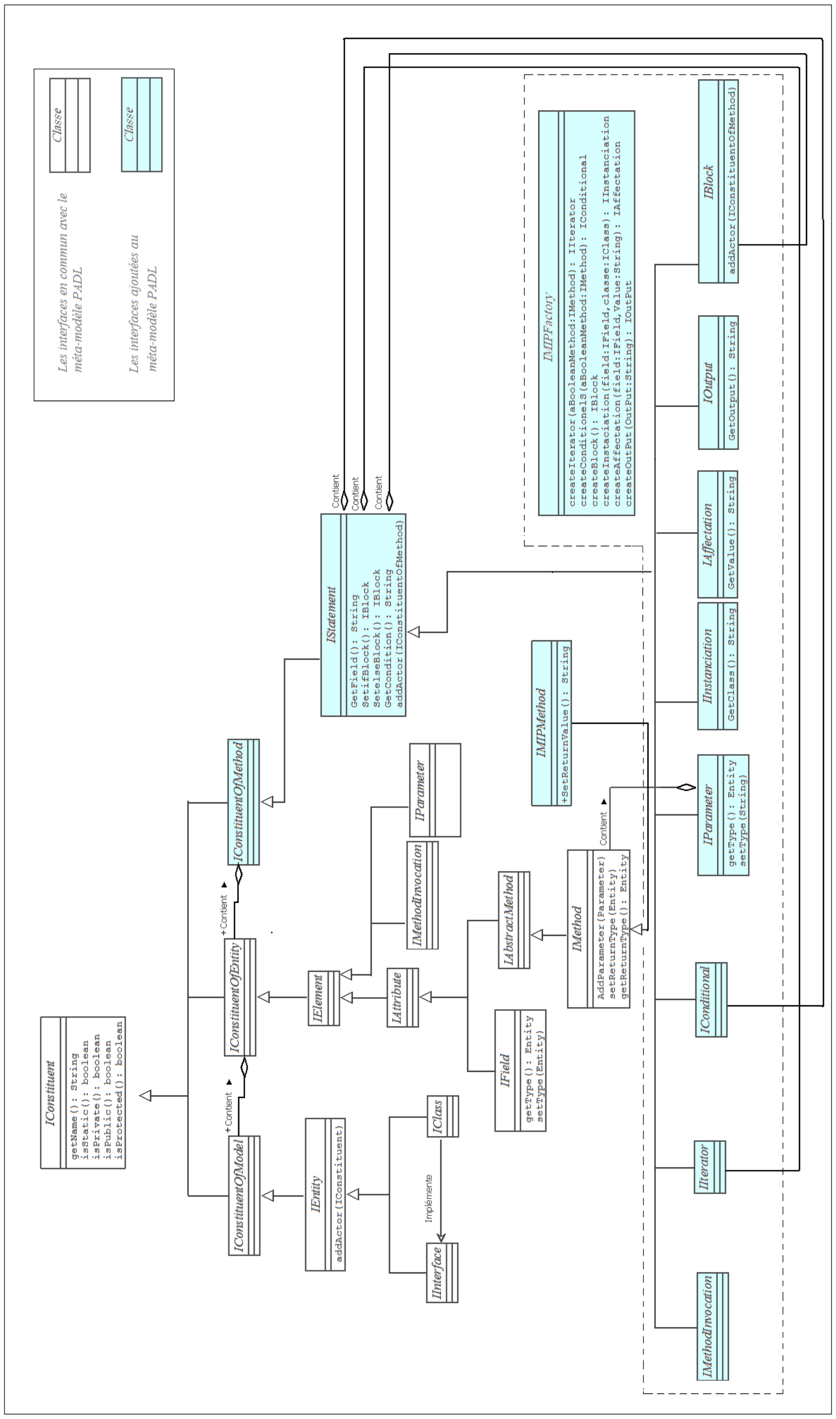


FIGURE 4.9: The MIP extension to the PADL meta-model.

10

References

[1] ASML Home Page, www.research.microsoft.com/foundations/AsmL/.

[2] C# language and tools:
http://msdn.microsoft.com/en-us/vcsharp/aa336809.aspx.

[3] DotGNU: http://www.gnu.org/software/dotgnu/.

[4] ECMA-335: Common language specification. http://www.ecma-international.org/publications/standards/Ecma-335.htm.

[5] ECMA and ISO/IEC: C# and common language infrastructure standards. http://msdn.microsoft.com/en-us/netframework/aa569283.aspx.

[6] RapidIO Document Specifications. http://www.rapidio.org.

[7] SystemC version 2.1. http://www.systemc.org/.

[8] Xilinx EDK: http://www.xilinx.com/ise/embedded/edk_docs.htm.

[9] *IEEE Standard VHDL Language Reference Manual*. IEEE, 1076, 2000 edition, 2000.

[10] AMBA Specification (rev2.0) and Multi layer AHB Specification, 2001.

[11] .NET source code. http://www.microsoft.com/net, 2003.

[12] OMG, UML Profile for Schedulability, Performance, and Time Specification. In *Version 1.0, http://www.omg.org*, 2003.

[13] ESys.NET. http://www.esys-net.org/, 2004.

[14] PROMPT-MAME Project Website. http://www.ele.etsmtl.ca/projets/PROMPT, 2005.

[15] *SystemC Language Reference Manual, IEEE Std 1666-2005*. 2005.

[16] QuickGraph, Graph Data Structures And Algorithms for .NET, 2008.

[17] Postsharp, 2009. http://www.postsharp.org/.

[18] Ben Albahari. A Comparative Overview of C#. http://genamics.com/developer/csharp_comparative.htm.

[19] Perry Alexander. Rosetta: Standardization at the System Level. *Computer*, 42(1):108–110, 2009.

[20] Dean Allemang and James A. Hendler. *Semantic Web for the Working Ontologist: Modeling in RDF, RDFS and OWL.* Morgan Kaufmann Publishers/Elsevier, Amsterdam; Boston, 2008.

[21] Daniel Amyot, Luigi Logrippo, Raymond J. A. Buhr, and Tom Gray. Use Case Maps for the Capture and Validation of Distributed Systems Requirements. In *RE '99: Proceedings of the 4th IEEE International Symposium on Requirements Engineering*, page 44, Washington, DC, USA, 1999. IEEE Computer Society.

[22] L. Aqvist. *Introduction to Deontic Logic and the Theory of Normative Systems*. Bibliopolis, 1983.

[23] Pontus Aström, Stefan Johansson, and Peter Nilsson. Application of Software Design Datterns to DSP Library Design. In *14th International Symposium on System Synthesis*, Montréal, Québec, Canada, 2001.

[24] Ivan Augé, Frédéric Pétrot, and Denis Hommais. A Pragmatic Approach To The Design of Embedded Systems. In *DATE'01: Proc. of Design Automation and Test in Europe*, pages 170–174, Munich, Germany, March 2001. IEEE.

[25] Jean Bacon. *Operating Systems: Concurrent and Distributed Software Design*. Addison-Wesley, Boston, 2003.

[26] Christopher J.O. Baker and Kei-Hoi Cheung, editors. *Semantic Web : Revolutionizing Knowledge Discovery in the Life Sciences*. Springer, 2007.

[27] Felice Balarin, Yosinori Watanabe, Harry Hsieh, Luciano Lavagno, Claudio Passerone, and Alberto Sangiovanni-Vincentelli. Metropolis: An Integrated Electronic System Design Environment. *Computer*, 36(4):45–52, 2003.

[28] K. Suzanne Barber, Thomas J. Graser, Jim Holt, and Geoff Baker. Arcade: Early Dynamic Property Evaluation of Requirements Using Partitioned Software Architecture Models. *Requirements Engineering*, 8(4):222–235, 2003.

[29] Kent Beck and Ralph E. Johnson. Patterns Generate Architectures. In *Proceedings of* 8^{th} *European Conference for Object-Oriented Programming*, pages 139–149. Springer-Verlag, July 1994.

[30] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the USENIX Annual Technical Conference*, pages 41–46. USENIX Association, 2005.

[31] Claude Berge. *Graphes et hypergraphes (in French)*, chapter 2, page 26. Dunod, 2nd edition, 1973.

[32] Janick Bergeron. *Writing Testbenches: Functional Verification of HDL Models, Second Edition*. Kluwer Academic Publishers, Norwell, MA, USA, 2003.

[33] Anders Berglund, Scott Boag, Don Chamberlin, Mary F. Fernndez, Mikael Kay, Jonathan Robie, and Jérome Siméon. XML Path Language (XPath) 2.0. Technical report, 23 January 2007.

[34] Tim Berners-Lee. N3 Notation: http://www.w3.org/DesignIssues/Notation3 .

[35] Tim Berners-Lee, James A. Hendler, and Ora Lassila. The Semantic Web. *Scientific American*, 284(5):28–37, 2001.

[36] J. Bhasker. *A SystemC Primer*. Star Galaxy, 2004.

[37] Scott Boag, Don Chamberlin, Mary F. Fernndez, Daniela Florescu, Jonathan Robie, and Jérome Siméon. XQuery 1.0: An XML Query Language. Technical report, W3C Recommendation - http://www.w3.org/TR/xquery/, 23 January 2007.

[38] Grady Booch, James Rumbaugh, and Ivar Jacobson. *Unified Modeling Language User Guide, The (2nd Edition) (Addison-Wesley Object Technology Series)*. Addison-Wesley Professional, 2005.

[39] Aimen Bouchhima, Iuliana Bacivarov, Wassim Youssef, Marius Bonaciu, and Ahmed A. Jerraya. Using Abstract CPU Subsystem Simulation Model for High Level HW/SW Architecture Exploration. In *ASPDAC'05: Proc. of the Asia South Pacific Design Automation Conference*, pages 969 – 972, 2005.

[40] Aimen Bouchhima, Patrice Gerin, and Frédéric Pétrot. Automatic Instrumentation of Embedded Software for High Level Hardware/Software Co-Simulation. In *ASP-DAC'09: Proc. of the Asia and South Pacific Design Automation Conference*, pages 546–551, Piscataway, NJ, USA, 2009. IEEE Press.

[41] Aimen Bouchhima, Sungjoo Yoo, and Ahmed Jerraya. Fast and Accurate Timed Execution of High Level Embedded Software using HW/SW Interface Simulation Model. In *ASPDAC'04: Proc. of the Asia South Pacific Design Automation Conference*, pages 469 – 474, 2004.

[42] Don Box. *Essential COM*. Addison-Wesley Professional, first edition, 1998.

[43] Robert King Brayton, Alberto L. Sangiovanni-Vincentelli, Curtis T. McMullen, and Gary D. Hachtel. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers, Norwell, MA, USA, 1984.

[44] Richard Buchmann, Alain Greiner, and Frédéric Pétrot. Fast Cycle Accurate Simulator To Simulate Event-Driven Behavior. In *In Proc. of the International Conference on Electrical Electronic and Computer Engineering*, pages 37–40, Cairo, Egypt, September 2004.

[45] Jerry Burch, Roberto Passerone, and Alberto L. Sangiovanni-Vincentelli. Overcoming Heterophobia: Modeling Concurrency in Heterogeneous Systems. In *ACSD '01: Proceedings of the Second International Conference on Application of Concurrency to System Design*, page 13, 2001.

[46] Timothy M. Burks and Karem A. Sakallah. Min-max Linear Programming and the Timing Analysis of Digital Circuits. In *ICCAD'93: Proc. of the International Conference on Computer-Aided Design*, pages 152–155, 1993.

[47] Joáo M. P. Cardoso and Horácio C. Neto. Compilation for FPGA-Based Reconfigurable Hardware. *IEEE Design and Test*, 20(2):65–75, 2003.

[48] Jeremy J. Carroll, Ian Dickinson, Chris Dollin, Dave Reynolds, Andy Seaborne, and Kevin Wilkinson. Jena: Implementing the Semantic Web Recommendations. In *WWW Alt. '04: Proceedings of the 13th international World Wide Web conference on Alternate track papers and posters*, pages 74–83, New York, NY, USA, 2004. ACM.

[49] Calin Cascaval, Colin Blundell, Maged Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. Software Transactional Memory: Why is it only a Research Toy?, url = http://dx.doi.org/10.1145/1400214.1400228, volume = 51, year = 2008. *Commun. ACM*, (11):40–46.

[50] Wander O. Cesario, Gabriela Nicolescu, Lovic Gauthier, Damien Lyonnard, and Ahmed A. Jerraya. Colif: A Design Representation for Application-Specific Multiprocessor SOCs. *Design and Test of Computers, IEEE*, 18(5):8–20, Sept-Oct 2001.

[51] Luc Charest, Michel Reid, El Mostapha Aboulhamid, and Guy Bois. A Methodology for Interfacing Open Source SystemC with a Third Party Software. In *DATE'01: Proceedings of the Design Automation and Test in Europe Conference*, pages 16–20, Munich, Germany, March 2001. IEEE Computer Society.

[52] Yiping Cheng and Da-Zhong Zheng. Min-Max Inequalities and the Timing Verification Problem with Max and Linear Constraints. *Discrete Event Dynamic Systems*, 15(2):119–143, 2005.

[53] Yiping Cheng and Da-Zhong Zheng. An Algorithm for Timing Verification of Systems Constrained by Min—max Inequalities. *Discrete Event Dynamic Systems*, 17(1):99–129, 2007.

[54] Nicos Christofides. *Graph Theory, An Algorithmic Approach*, chapter 10, Hamiltonian Circuits, Paths and the Traveling Salesman Problem, pages 214–235. Academic Press, 1975.

[55] Alexandre Chureau, Yvon Savaria, and El Mostapha Aboulhamid. The Role of Model-Level Transactors and UML in Functional Prototyping of Systems-on-Chip: A Software-Radio Application. In *DATE'05: Proc. of the conference on Design, Automation and Test in Europe*, pages 698–703, 2005.

[56] Kendall Grant Clark, Lee Feigenbaum, and Elias Torres. SPARQL Protocol for RDF. Technical report, W3C Recommendation, 15 January 2008.

[57] W.F. Clocksin and C.S. Mellish. *Programming in Prolog*. Springer-Verlag, 1987.

[58] Bob Cmelik and David Keppel. Shade: A Fast Instruction Set Simulator for Execution Profiling. In *Sigmetrics 94*, pages 128–138, June 1994.

[59] A. Colgan and P. Hardee. Advancing Transaction Level Modeling: Linking the OSCI and OCP-IP Worlds at Transaction Level, http://www.opensystems-publishing.com/whitepapers.

[60] C.T.I. Comete. *CODESIGN: Conception conjointe logiciel-matériel, in french*. Eyrolles, 1998.

[61] James Coplien, Daniel Hoffman, and David Weiss. Commonality and Variability in Software Engineering. *IEEE Software*, 15(6):37–45, 1998. James O. Coplien, Daniel M. Hoffman, and David M. Weiss. Commonality and Variability in Software Engineering.

[62] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Graph Algorithms*, chapter 23, pages 485–488. MIT Press, 2nd printing, 1994.

[63] György Csertán, Gábor Huszerl, István Majzik, Zsigmond Pap, András Pataricza, and Dániel Varró. VIATRA ”Visual Automated Transformations for Formal Verification and Validation of UML Models. In *ASE ’02: Proceedings of the 17th IEEE International Conference on Automated Eoftware Engineering*, page 267, 2002.

[64] Robertas Damaševičius, Giedrius Majauskas, and Vytautas Štuikys. Application of Design Patterns for Hardware Design. In *DAC’03, proc. of the 40th International Design Automation Conference*, pages 48–53, Anaheim, California, USA, June 2003. ACM Press.

[65] Chris J. Date. *An Introduction to Database Systems 7ed.* Addison Wesley Longman, 2000.

[66] Stefan Decker, Sergey Melnik, Frank Van Harmelen, Dieter Fensel, Michel Klein, Jeen Broekstra, Michael Erdmann, and Ian Horrocks. The Semantic Web: the Roles of XML and RDF. *Internet Computing, IEEE*, 15(3):63–74, Sept-Oct 2000.

[67] Fredrik Degerlund, Marina Walden, and Kaisa Sere. Implementation Issues Concerning the Action Systems Formalism. In *PDCAT ’07: Proceedings of the Eighth International Conference on Parallel and Distributed Computing, Applications and Technologies*, pages 471–479, Washington, DC, USA, 2007. IEEE Computer Society.

[68] Louise A. Dennis, Graham Collins, Michael Norrish, Richard J. Boulton, Konrad Slind, Graham Robinson, Michael J. C. Gordon, and Thomas F. Melham. The PROSPER Toolkit. In *International Journal on Software Tools for Technology Transfer vol. 4, n. 2*, 2003.

[69] Paolo Destro, Franco Fummi, and Graziano Pravadelli. A Smooth Refinement Flow for Co-Designing HW and SW Threads. In *DATE'07: Proc. of the conference on Design, Automation and Test in Europe*, pages 105–110, 2007.

[70] Adam Donlin. Transaction level modeling: flows and use models. In *CODES+ISSS '04: Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 75–80, New York, NY, USA, 2004. ACM.

[71] Frédéric Doucet, Sandeep Shukla, and Rajesh Gupta. Introspection in System-Level Language Frameworks: Meta-level vs. Integrated. In *DATE'03: Design Automation and Test in Europe Conference*, pages 382–387, Munich, Germany, 2003. IEEE Computer Society.

[72] Mathieu Dubois and El Mostapha Aboulhamid. Techniques to Improve Cosimulation and Interoperability of Heterogeneous Models. *Electronics, Circuits and Systems, 2005. ICECS 2005. 12th IEEE International Conference on*, pages 1–4, Dec. 2005.

[73] Mathieu Dubois, El Mostapha Aboulhamid, and Frédéric Rousseau. Acceleration for a Compiled Transaction Level Modeling Simulation. *Electronics, Circuits and Systems, 2006. ICECS '06. 13th IEEE International Conference on*, pages 1176–1179, Dec. 2006.

[74] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in Property Specifications for Finite-State Verification. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 411–420, New York, NY, USA, 1999. ACM.

[75] ”eCosCentric”. eCos homepage, http://ecos.sourceware.org/.

[76] Stephen Edwards, Luciano Lavagno, Edward A. Lee, and Alberto Sangiovanni-Vincentelli. Design of Embedded Systems: Formal Models, Validation, and Synthesis. *Proc. of the IEEE*, 85(3):366–390, March 1997.

[77] Cindy Eisner, Dana Fisman, John Havlicek, Yoad Lustig, Anthony McIsaac, and David Van Campenhout. Reasoning with Temporal Logic on Truncated Paths. In *15th International Conference on Computer Aided Verification*, 2003.

[78] A. El-Aboudi and El Mostapha Aboulhamid. An Algorithm for the Verification of Timing Diagrams Realizability. In *ISCAS (1)*, pages 314–317, 1999.

[79] A. El-Aboudi, El Mostapha Aboulhamid, and Eduard Cerny. Verificatiom of Synchronous Realizability of Interfaces from Timing Diagram Specifications. *Microelectronics, 1998. ICM '98. Proceedings of the Tenth International Conference on*, pages 103–106, 1998.

[80] Jan Ellsberger, Dieter Hogrefe, and Amardeo Sarma. *SDL: Formal Object-oriented Language for Communicating Systems*. Prentice Hall, 1997.

[81] B. Berar et al. *Systems and Software Verification, Model-Checking Techniques and Tools*. Springer-Verlag, 2001.

[82] Alessandro Fantechi, Stefania Gnesi, G. Lami, and A. Maccari. Application of Linguistic Techniques for Use Case Analysis. In *RE '02: Proceedings of the 10th Anniversary IEEE Joint International Conference on Requirements Engineering*, pages 157–164, Washington, DC, USA, 2002. IEEE Computer Society.

[83] International Technology Roadmap for Semiconductor. 2004 Edition. In *http://public.itrs.net/*, 2004.

[84] Christopher P. Fuhrman. Lightweight Models for Interpreting Informal Specifications. *Requirements Engineering*, 8(4):206–221, 2003.

[85] Ariel Fuxman, Lin Liu, John Mylopoulos, Marco Roveri, and Paolo Traverso. Specifying and Analyzing Early Requirements in Tropos. *Requirements Engineering*, 9(2):132–150, 2004.

[86] Anthony Joseph Gahlinger. *Coherence and Satisfiability of Waveform Timing Specifications*. PhD thesis, Waterloo, Ont., Canada, Canada, 1990.

[87] Daniel D. Gajski, Frank Vahid, Sanjiv Narayan, and Jie Gong. *Specification and Design of Embedded Systems*. Prentice Hall, 1994.

[88] Daniel D. Gajski, Jianwen Zhu, Rainer Dömer, Andreas Gerstlauer, and Shuquing Zhao. *SpecC: Specification Language and Methodology*. Kluwer Academic Publishers, Boston, March 2000.

[89] Erich Gamma, Richard Helm, Ralph Johnso, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.

[90] Paul Gastin and Denis Oddoux. Fast LTL to Buchi Automata Translation. In *CAV '01: Proceedings of the 13th International Conference on Computer Aided Verification*, pages 53–65, London, UK, 2001. Springer-Verlag.

[91] Patrice Gerin, Hao Shen, Alexandre Chureau, and Ahmed Jerraya. Flexible and Executable Hardware/Software Interface Modeling for Multiprocessor SoC Design Using SystemC. In *ASPDAC'07: Proc. of the Asia South Pacific Design Automation Conference*, pages 390–395, 2007.

[92] Andreas Gerstlauer, Haobo Yu, and Daniel D. Gajski. RTOS Modeling for System Level Design. In *DATE'03: Proc. of the Design Automation and Test in Europe Conference*, pages 130–135, 2003.

[93] Rob Gerth, Doron Peled, Moshe Y. Vardi, and Pierre Wolper. Simple on-the-fly Automatic Verification of Linear Temporal Logic. In *Proceedings of the Fifteenth IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification XV*, pages 3–18, London, UK, UK, 1995. Chapman & Hall, Ltd.

[94] Dimitra Giannakopoulou and Klaus Havelund. Automata-Based Verification of Temporal Properties on Running Programs. In *ASE '01: Proceedings of the 16th IEEE international conference on Automated software engineering*, page 412, Washington, DC, USA, 2001. IEEE Computer Society.

[95] Bruno Girodias, El Mostapha Aboulhamid, and Gabriela Nicolescu. A Platform for Refinement of OS Services for Embedded Systems. In *DELTA '06: Proceedings of the Third IEEE International Workshop on Electronic Design, Test and Applications*, pages 227–236, 2006.

[96] Maya B. Gokhale and Janice M. Stone. NAPA C: Compiling for a Hybrid RISC/FPGA Architecture. In *FCCM '98: Proc. of the IEEE Symposium on FPGAs for Custom Computing Machines*, page 126, 1998.

[97] Nicolas Gorse, Pascale Bélanger, El Mostapha Aboulhamid, and Yvon Savaria. Mixing Linguistic and Formal Techniques for High-Level Requirements Engineering. *Proceedings of the 16th IEEE International Conference on Microelectronics, Tunisia*, 2004.

[98] Nicolas Gorse, Pascale Bélanger, Alexandre Chureau, El Mostapha Aboulhamid, and Yvon Savaria. A high-Level Requirements Engineering Methodology for Electronic System-Level Design. *Comput. Electr. Eng.*, 33(4):249–268, 2007.

[99] K John Gough. Stacking them up: a Comparison of Virtual Machines. *Australasian Computer Science Communnication*, 23(4):55–61, 2001.

[100] Thorsten Grotker, Stan Liao, Grant Martin, and Stuart Swan. *System Design with SystemC*. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[101] Michael Grove and Andrew Schain. POPS NASAs Expertise Location Service Powered by Semantic Web Technologies. Technical report, W3C Semantic Web Case Studies and Use Cases - http://www.w3.org/2001/sw/sweo/public/UseCases/Nasa/Nasa.pdf, 2008.

[102] Yann-Gaël Guéhéneuc and Hervé Albin-Amiot. Recovering Binary Class Relationships: Putting Icing on the UML Cake. In Doug C. Schmidt, editor, *Proceedings of the* 19^{th} *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 301–314. ACM Press, October 2004.

[103] Yann-Gaël Guéhéneuc and Giuliano Antoniol. DeMIMA: A Multi-layered Framework for Design Pattern Identification. *Transactions on Software Engineering*, 34(5):667–684, September 2008.

[104] Rachid Guerraoui, Maurice Herlihy, and Bastian Pochon. Polymorphic Contention Management. In *DISC '05: Proceedings of the nineteenth International Symposium on Distributed Computing*, pages 303–323. LNCS, Springer, Sep 2005.

[105] Elliotte R. Harold and Scott W. Means. *XML in a Nutshell, Third Edition*. O'Reilly Media, Inc., October 2004.

[106] Tim Harris and Keir Fraser. Language Support for Lightweight Transactions. In *OOPSLA'03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 388–402, New York, NY, USA, 2003. ACM.

[107] Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy. Composable Memory Transactions. *Commun. ACM*, 51(8):91–100, 2008.

[108] Abdelsalam Hassan, Keishi Sakanushi, Yoshinori Takeuchi, and Masaharu Imai. RTK-Spec TRON: A Simulation Model of an ITRON Based RTOS Kernel in SystemC. In *DATE'05: Proc. of the Design Automation and Test in Europe Conference*, pages 554–559, 2005.

[109] Claude Helmstetter, Florence Maraninchi, Laurent Maillet-Contoz, and Matthieu Moy. Automatic generation of schedulings for improving the test coverage of systems-on-a-chip. In *FMCAD '06: Proceedings of the Formal Methods in Computer Aided Design*, pages 171–178, Washington, DC, USA, 2006. IEEE Computer Society.

[110] John L. Hennessy and David A. Patterson. *Computer Architecture, a Quantitative Approach*. Morgan Kaufmann Publisher, Inc, 1990.

[111] John L. Hennessy and David A. Patterson. *Computer Architecture and Design, The Hardware/Software Interface*. Morgan Kaufmann Publisher, Inc, 2003.

[112] Maurice Herlihy. Obstruction-free Synchronization: Double-ended Queues as an Example. In *In Proceedings of the 23rd International Conference on Distributed Computing Systems*, pages 522–529. IEEE Computer Society, 2003.

[113] Patrick Heymans and Eric Dubois. Scenario-Based Techniques for Supporting the Elaboration and the Validation of Formal Requirements. *Requirements Engineering*, 3(3/4):202–218, 1998.

[114] R. Hilderink and T. Grötker. Transaction-level Modeling of Bus-based Systems with SystemC 2.0. Synopsys, Inc.

[115] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[116] C. A. R. Hoare. Towards a Theory of Parallel Programming. pages 231–244, 2002.

[117] A. Horn. On Sentences Which are True of Direct Unions of Algebras. *Journal of symbolic logic*, pages 14–21, 1951.

[118] Ian Horrocks, Peter F. Patel-Schneider, Harold Boley, Said Tabet, Benjamin Grosof, and Mike Dean. SWRL: A Semantic Web Rule Language Combining OWL and RuleML. Technical report, W3C Member Submission - http://www.w3.org/Submission/2004/SUBM-SWRL-20040521/, 21 May 2004.

[119] Intel. IXP45X datasheet, http://www.intel.com/design/network/datashts/306261.htm.

[120] ITRS. International technology roadmap for semiconductors design, 2007. http://www.itrs.net/Links/2007ITRS/2007_Chapters/2007_Design.pdf.

[121] ITU. *Recommendation Z.120: Message Sequence Chart (MSC)*. 1996.

[122] Glenn Jennings. A Case Against Event Driven Simulation for Digital System Design. In *24th Annual Simulation Symposium*, pages 170–175, April 1991.

[123] Glenn Jennings. A Case Against Event-driven Simulation for Digital System Design . In *Simulation Symposium, 1991. Proceedings of the 24th Annual*, pages 170–176, 1991.

[124] Weixing Ji, Feng Shi, and Baojun Qiao. The Design of a Novel Object Processor: OOMIPS. In *Proceedings of the 18th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP 2007)*, July 2007.

[125] Viraj Kamat. Towards slicing vhdl. Master's thesis, Indian Institute of Technology, Bombay, 2003.

[126] Torsten Kempf, Kingshuk Karuri, Stefan Wallentowitz, Gerd Ascheid, Rainer Leupers, and Heinrich Meyr. A SW Performance Estimation Framework for Early System-Level-Design using Fine-Grained Instrumentation. In *DATE'06: Proc. of the Design Automation and Test in Europe Conference*, pages 468–473, 2006.

[127] K. Khordoc and E. Cerny. Semantics and Verification of Action Diagrams with Linear Timing. *ACM Trans. Des. Autom. Electron. Syst.*, 3(1):21–50, 1998.

[128] Albert Carl Jan KIENHUIS Kienhuis. *Design Space Exploration of Stream-based Dataflow Architctures: Methods and Tools*. PhD thesis, Delft University of Technology, 1999.

[129] Holger Knublauch, Mark A Musen, and Alan L Rector. Editing Description Logic Ontologies with the Protege OWL Plugin. In *In Description Logics*, 2004.

[130] Donald E. Knuth. *The Stanford GraphBase*, chapter Roget Components, pages 512–519. Addison Wesley Publishing Company, 1994.

[131] Cedric Koch-Hofer, Marc Renaudin, Yvain Thonnart, and Pascal Vivet. ASC, a SystemC Extension for Modeling Asynchronous Systems, and Its Application to an Asynchronous NoC. In *NOCS '07: Proc. of the First International Symposium on Networks-on-Chip*, pages 295–306, Washington, DC, USA, 2007. IEEE Computer Society.

[132] Thomas Kropf. *Introduction to Formal Hardware Verification: Methods and Tools for Designing Correct Circuits and Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[133] Wido Kruijtzer, Pieter van der Wolf, Erwin de Kock, Jan Stuyt, Wolfgang Ecker, Albrecht Mayer, Serge Hustin, Christophe Amerijckx, Serge de Paoli, and Emmanuel Vaumorin. Industrial IP Integration Flows Based on IP-XACT$^{\text{TM}}$Standards. In *DATE'08: Proc. of the conference on Design, automation and test in Europe*, pages 32–37, New York, NY, USA, 2008. ACM.

[134] Marcello Lajolo, Mihai Lazarescu, and Alberto Sangiovanni-Vincentelli. A Compilation-Based Software Estimation Scheme for Hardware/Software Co-Simulation. In *Proc. of the International Workshop on Hardware/Software Codesign*, pages 85–89, May 1999.

[135] J. Lapalme, E.M. Aboulhamid, G. Nicolescu, L. Charest, F.R. Boyer, J.P. David, and G. Bois. .NET Framework - a Solution for the Next Generation Tools for System-Level Modeling and Simulation. In *DATE'04: Proc. of the Design Automation and Test in Europe Conference*, volume 1, pages 732–733, 2004.

[136] James Lapalme. Esys.net : a new .net based system-level design environment. Master's thesis, Universite de Montreal, 2003.

[137] James Lapalme, El Mostapha Aboulhamid, and Gabriela Nicolescu. A new efficient EDA tool design methodology. *ACM Transaction on Embedded Computing Systems*, 5(2):408–430, 2006.

[138] James Lapalme, El Mostapha Aboulhamid, Gabriela Nicolescu, Luc Charest, Franois R. Boyer, Jean Pierre David, and Guy Bois. Esys.NET: a New Solution for Embedded Systems Modeling and Simulation. *SIGPLAN Not.*, 39(7):107–114, 2004.

[139] James Lapalme, El Mostapha Aboulhamid, Gabriela Nicolescu, Luc Charest, Franois R. Boyer, Jean Pierre David, and Guy Bois. .NET Framework - a Solution for the Next Generation Tools for System-Level Modeling and Simulation. *DATE'04: Proc. of Design Automation and Test in Europe Conference*, 1:732–733, Feb. 2004.

[140] James Lapalme, El Mostapha Aboulhamid, Gabriela Nicolescu, and Frédéric Rousseau. Separating Modeling and Simulation Aspects in Hardware/Software System Design. *Microelectronics, 2006. ICM '06. International Conference on*, pages 202–205, Dec. 2006.

[141] James Larus and Christos Kozyrakis. Transactional Memory. *Commun. ACM*, 51(7):80–88, 2008.

[142] James R. Larus and Ravi Rajwar. *Transactional Memory*. Morgan & Claypool, 2006.

[143] Edward A. Lee and Stephen Neuendorffer. MoML A Modeling Markup Language in XML, Version 0.4. Technical Report ERL/UCB M 00/12, University of California at Berkeley,, 2000.

[144] Jesse Liberty. Programming C#: Attributes and Reflection, O'Reilly, http://www.ondotnet.com/pub/a/dotnet/excerpt/prog_csharp_ch18/index.html, 2001.

[145] D. B. Lomet. Process Structuring, Synchronization, and Recovery Using Atomic Actions. In *Proceedings of an ACM conference on Language design for reliable software*, pages 128–137, 1977.

[146] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. 1995.

[147] Grant Martin. Systemc tools. In *European SystemC Users Group Meeting*.

[148] Kenneth L. McMillan and David L. Dill. Algorithms for Interface Timing Verification. In *ICCD '92: Proc. of the IEEE International Conference on Computer Design on VLSI in Computer & Processors*, pages 48–51, 1992.

[149] Nenad Medvidovic and Richard N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, 2000.

[150] A. Mel'cuk. Dependency in Linguistic Description. http://www.olst.umontreal.ca/FrEng/Dependency.pdf.

[151] Giovanni De Micheli. *Synthesis and Optomization of Digital Circuits*. McGraw-Hill, USA, 1994.

[152] Rocco Le Moigne, Olivier Pasquier, and Jean-Paul Calvez. A Generic RTOS Model for Real-time Systems Simulation with SystemC. In *DATE'04: Proc. of the Design Automation and Test in Europe Conference*, pages 82–87, 2004.

[153] Matthieu Moy, Florence Maraninchi, and Laurent Maillet-Contoz. Pinapa: an Extraction Tool for SystemC Descriptions of Systems-on-a-Chip. In *EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software*, pages 317–324, New York, NY, USA, 2005. ACM.

[154] José M. Moya, Fernando Rincón, Francisco Moya, and Juan Carlos López. Improving Embedded System Design by Means of HW-SW Compilation on Reconfigurable Coprocessors. In *ISSS'02: Proceedings of the 15th international Symposium on System Synthesis*, pages 255–260, 2002.

[155] MSDN. Dynamic-link libraries, http://msdn.microsoft.com.

[156] Eric K. Neumann and Dennis Quan. Biodash: A Semantic Web Dashboard for Drug Development. In *Pacific Symposium on Biocomputing*, pages 176–187, 2006.

[157] James Newkirk and Alexei A. Vorontsov. How .NET's Custom Attributes Affect Design. *IEEE Software*, 19(5):18–20, 2002.

[158] Ralf Niemann. *Hardware/Software Co-Design for Data Flow Dominated Embedded Systems*. Kluwer Academic Publishers, Norwell, MA, USA, 1998.

[159] Open SystemC Initiative (OSCI). Functional specification for SystemC 2.0, 2001. http://www.systemc.org.

[160] Open SystemC Initiative (OSCI). SystemC user guide, 2001. http://www.systemc.org.

[161] Samir Palnitkar. *Verilog®HDL: a Guide to Digital Design and Synthesis, second edition*. Prentice Hall Press, Upper Saddle River, NJ, USA, 2003.

[162] T. Parr. Stringtemplate documentation, http://www.antlr.org/stringtemplate/index.tml, 2003.

[163] Terence Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Programmers. Pragmatic Bookshelf, first edition, May 2007.

[164] Claudio Passerone, Massimiliano Chiodo, Wilsin Gosti, Luciano Lavagno, and Alberto Sangiovanni-Vincentelli. Evaluation of Trade-offs in the Design of Embedded Systems via Co-Simulation. Technical Report UCB-ERL-96-12, University of California, Berkeley, Computer Science Department, University of California, Berkeley, 1996.

[165] H. D. Patel, D A. Mathaikutty, D. Berner, and S. K. Shukla. SystemCXML: An extensible systemc front end using XML, http://systemcxml.sourceforge.net/. http://systemcxml.sourceforge.net/, 2005.

[166] Hiren D. Patel and Sandeep K. Shukla. Tackling an abstraction gap: co-simulating SystemC DE with bluespec ESL. In *DATE'07: Proceedings of the conference on Design, automation and test in Europe*, pages 279–284, San Jose, CA, USA, 2007. EDA Consortium.

[167] James A. Payne. *Introduction to Simulation: Programming Techniques and Methods of Analysis*, chapter 2, pages 11–22. McGraw-Hill, 1982.

[168] F. Pereira and D.H.D. Warren. Definite Clause Grammars for Language Analysis - a Survey of the Formalism and a Comparison with Augmented Transition Networks. In *Artificial Intelligence Journal, Vol. 13*, 1980.

[169] Hector G. Perez-Gonzalez and Jugal K. Kalita. GOOAL: a Graphic Object Oriented Analysis Laboratory. In *OOPSLA '02: Companion of the 17th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 38–39, New York, NY, USA, 2002. ACM.

[170] Frédéric Pétrot, Denis Hommais, and Alain Greiner. A Simulation Environment for Core Based Embedded Systems. In *Proc. of the 30th Int. Simulation Symp.*, pages 86–91, Atlanta, Georgia, April 1997.

[171] Frédéric Pétrot, Denis Hommais, and Alain Greiner. Cycle Precise Core Based Hardware/Software System Simulation with Predictable Event Propagation. In *Proceeding of the 23rd Euromicro Conference*, pages 182–187, Budapest, Hungary, September 1997. IEEE.

[172] A. Pnueli. The Temporal Logic of Programs. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*, pages 46 – 67, 1977.

[173] Hector Posadas, Jesús Ádamez, Pablo Sánchez, Eugenio Villar, and Francisco Blasco. POSIX Modeling in SystemC. In *ASPDAC'05: Proc. of the Asia South Pacific Design Automation Conference*, pages 485–490, 2006.

[174] E. Prudhommeaux and A. Seaborne. SPARQL Query Language for RDF. Technical Report REC-rdf-schema-20040210, World Wide Web Consortium, Jan. 2008.

[175] Kaye R. Seamless with C-bridge: C Based Co-Verification. In *Technical Papers, Mentor*, page 27, 2002.

[176] H. Reichel R. Deutschmann, M. Fruth and H.-C. Reuss. Trace Checking with Real-Time Specifications. In *5th Symposium on Formal Methods for Automation and Safety in Railway and Automotive Systems*, 2004.

[177] Debbie Richards. Merging Individual Conceptual Models of Requirements. *Requir. Eng.*, 8(4):195–205, 2003.

[178] D. F. Robinson and L. R. Foulds. Acyclic digraphs *in Digraphs: Theory and Techniques*, chapter 3.6, pages 86–90. Gordon and Breach Scientific Publishers, 1980.

[179] D. F. Robinson and L. R. Foulds. Digraph structure*, in Digraphs: Theory and Techniques*, chapter 2, pages 43–62. Gordon and Breach Scientific Publishers, 1980.

[180] Bertil Roslund and Per Andersson. A Flexible Technique for OS-Support in Instruction Level Simulators. In *27th Annual Simulation Symposium*, pages 134–141, La Jolla, CA, April 1994. SCS, IEEE Press.

[181] Luc Séméria, Koichi Sato, and Giovanni De Micheli. Synthesis of Hardware Models in C with Pointers and Complex Data Structures. *IEEE Transactions on VLSI Systems*, 9(6):743–756, 2001.

[182] Wuwei Shen, Kevin Compton, and James Huggins. A UML Validation Toolset Based on Abstract State Machines. In *ASE '01: Proceedings of the 16th IEEE international conference on Automated software engineering*, page 315, Washington, DC, USA, 2001. IEEE Computer Society.

[183] Evren Sirin, Bijan Parsia, Bernardo Cuenca Grau, Aditya Kalyanpur, and Yarden Katz. Pellet: A Practical OWL-DL Reasoner. *Web Semantics: Science, Services and Agents on the World Wide Web*, 5(2):51–53, 2007.

[184] S. Smith, M. Ray Mercer, and B. Brock. Demand Driven Simulation: BACK-SIM. In *DAC'87: 24st Design Automation Conference*, pages 181–187, San Diego, CA, June 1987. ACM/IEEE, IEEE Press.

[185] S. Stuijk. *Predictable Mapping of Streaming Applications on Multiprocessors*. PhD thesis, Eindhoven University of Technology, The Netherlands, 2007.

[186] Ralf's Sudelbcher. Nst transactional memory, 2007. http://weblogs.asp.net/ralfw/archive/tags/Software+Transactional+Memory/default.aspx.

[187] Stuart Sutherland. SystemVerilog tutorial. http://www.systemverilog.org/techpapers/techpapers.html, 2003.

[188] Stuart Sutherland, Simon Davidmann, Peter Flake, and Phil Moorby. *System Verilog for Design: A Guide to Using System Verilog for Hardware Design and Modeling*. Kluwer Academic Publishers, Norwell, MA, USA, 2004.

[189] B. D. Theelen. Performance Model Generation for MPSoC Design-Space Exploration. In *QEST '08: Proceedings of the 2008 Fifth International Conference on Quantitative Evaluation of Systems*, pages 39–40, Washington, DC, USA, 2008. IEEE Computer Society.

[190] K. C. Thramboulidis, G. Doukas, and G. Koumoutsos. A SOA-based Embedded Systems Development Environment for Industrial Automation. *EURASIP J. Embedded Syst.*, 2008(1):1–15, 2008.

[191] Walter Tibboel, Victor Reyes, Martin Klompstra, and Dennis Alders. System-Level Design Flow Based on a Functional Reference for HW and SW. In *DAC'07: Proc. of the Design Automation Conference*, pages 23–28, 2007.

[192] Frank Vahid and Tony Givargis. *Embedded System Design: A Unified Hardware/Software Introduction*. John Wiley & Sons, Inc., New York, NY, USA, 2001.

[193] P.H.A. van der Putten and J.P.M. Voeten. *Specification of Reactive Hardware/Software Systems: The method Software/Hardware Engineering (SHE)*. Ph.d., Eindhoven University of Technology, 1997.

[194] Peter Vanbekbergen, Gert Goosens, and Hugo De Man. Specification and ananlysis of timing constraints in signal transition graph. In *DAC'92: Proceedings of the 29th annual conference on Design automation*, pages 302–306, New York, NY, USA, 1992. ACM.

[195] Emmanuel Viaud, Franģis Pêcheux, and Alain Greiner. An Efficient TLM/T Modeling and Simulation Environment Based on Conservative Parallel Discrete Event Principles. In *DATE'06: Proc. of the Design Automation and Test in Europe Conference*, pages 94–99, 2006.

[196] W3C. XSL Transformations (XSLT), 1999.

[197] Elizabeth A. Walkup and Gaetano Borriello. Interface Timing Verification with Application to Synthesis. In *DAC'94: Proceedings of the 31st annual conference on Design automation*, pages 106–112, New York, NY, USA, 1994. ACM.

[198] Gerhard Weikum and Gottfried Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[199] Wikipedia. Component Object Model — Wikipedia, The Free Encyclopedia.

[200] Wikipedia. .NET Reflector, 2008. http://en.wikipedia.org/wiki/.NET_Reflector.

[201] Wikipedia. Managed code — Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/wiki/Managed_code, 2009.

[202] Wikipedia. P/Invoke — Wikipedia, The Free Encyclopedia, 2009.

[203] Wikipedia. Virtual machine — Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/wiki/Virtual_machine, 2009.

[204] Pierre Wolper, Moshe Y. Vardi, and A. Prasad Sistla. Reasoning about Infin thite Computation Paths. In *Proceedings of the 24th IEEE Symposium on Foundations of Computer Science*, pages 185 – 194, 1983.

[205] World Wide Web Consortium (W3C). XML specification. http://www.w3c.org, 2003.

[206] Roel Wuyts. Declarative Reasoning About the Structure of Object-Oriented Systems. In Joseph Gil, editor, *Proceedings of the* 26^{th} *Conference on the Technology of Object-Oriented Languages and Systems*, pages 112–124. IEEE Computer Society Press, August 1998.

[207] Sudhakar Yalamanchili. *Introductory VHDL: From Simulation To Synthesis*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2004.

[208] Zhi Alex Ye, Nagaraj Shenoy, and Prithviraj Baneijee. A C Compiler for a Processor with a Reconfigurable Functional Unit. In *FPGA '00: Proc. of the 2000 ACM/SIGDA eighth international symposium on Field programmable gate arrays*, pages 95–100, 2000.

[209] Ti-Yen Yen, Alex Ishii, Al Casavant, and Wayne Wolf. Efficient algorithms for interface timing verification. *Form. Methods Syst. Des.*, 12(3):241–265, 1998.

[210] Ti-Yen Yen, Wayne Wolf, Al Casavant, and Alex Ishii. Efficient Algorithms for Interface Timing Verification. In *EURO-DAC'94: Proceedings of the conference on European design automation*, pages 34–39, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[211] Hong Zhu and Lingzi Jin. Scenario Analysis in an Automated Tool for Requirements Engineering. *Requirements Engineering*, 5(1):2–22, 2000.