Enhanced Reuse and Teamwork Capabilities for an Object-oriented Extension of VHDL

Michael Mrva
Siemens AG, Corporate Technology, ZT ME 5
D-81730 Munich, Germany
Michael.Mrva@mchp.siemens.de

Abstract

This paper presents a proposal for enabling VHDL to better support reuse and collaboration. Base idea is passing on the adequate information to partners working in an object-oriented hardware design environment. Appropriate subgoals for achieving this are:

- *an optimal mix of necessary abstraction and sufficient precision,*
- *a formal description consisting of implementation constraints and knowledge requirements, and*
- *the non-formal concept of mutual consideration.*

Several loans are made from

- *the software domain: Java interfaces, type models, and the request for habitability,*
- *the VHDL Annotation Language.*

This is not an experience report, for the idea of adopting the mentioned software concepts to hardware design is new. It is rather a guided tour to some "panorama views". Although they may not seem related to each other at first glance, they turn out to altogether support a common goal: understanding and communicating VHDL-based designs better.

0. Introduction

There are a number of object-oriented (OO) languages which do not distinguish between modeling and programming or only do so inadequately. C++ [22], for example, is one of these languages. There are other languages which in theory can make such a distinction but in practice they tend to blur the two processes because they state too many details for modeling. An example of the second type is, say, Eiffel [15]. Although, on one hand, it has a well thought out mechanism based on pre- and post conditions, referred to as "Programming by Contract" [23, 14], with which the role division between the supplier and user can be described, on the other hand, multiple inheritance for classes is permitted and the modeller must give a complete specification of the classes before compilation can take place.

(This article will not discuss Ada'95 any further; see eg. [2] for this language.)

The inability to distinguish between modeling and programming referred to above is a pity because it is precisely the principle of object orientation, rather than conventional techniques, that would provide the best basis for supporting teamwork and reuse if more attention were paid to the modeling phase. On the other hand, proposals for an OO extension of VHDL [19, 18], currently at the draft stage, have a good chance of turning out well in this respect because VHDL, the base language, is slanted towards modeling.

To prevent any misunderstandings, a fundamental difference should be made clear: While it should always be possible to run a *model* in a VHDL-based hardware design environment, this only rarely applies to the software technology. "Soft models" can only be run if there is a tool capable of running the model as a prototype for the selected model description language. Generally speaking, this is only the case in simulation or animation environments.

However, this differentiation does not bring us any further. The crucial principle for the effective support of teamwork and reuse - in both hardware and software design environments - is the provision of the right amount of information.

Two submechanisms ensure that an adequate amount of information is supplied: Abstraction and precision. There is nothing mutually exclusive about these objectives. I can be very precise at a relatively high degree of abstraction (see Fig. 1). Whereas abstraction means the deliberate omission of (currently) unneeded knowledge, it is the sufficiently exact formulation of essential knowledge that is crucial for the second art - precision - of effective communication between partners in a team. The same also applies to reuse: Here it is not just passing on important information to the participants in a single project, but the provision of potential participants in any future projects.

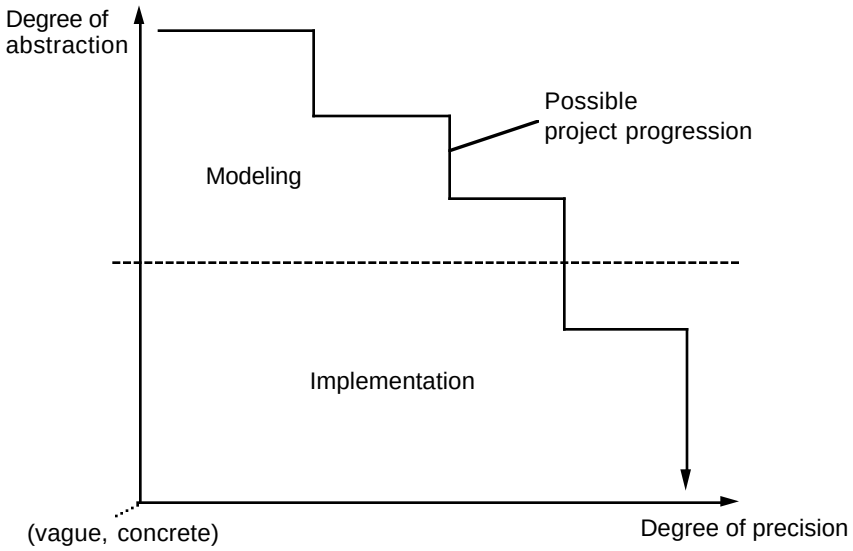

Fig. 1:
Coordinate system for one aspect of a HW/SW co-design and prototyping environment spanned by the dimensions precision (x) and abstraction (y)

1. The Java Interface Concept

A language in which the distinction between essential knowledge and (currently) unnecessary knowledge is almost built-in as a feature is Java [21]. This language is neither a superset or subset of C++, even if many constructs are strongly reminiscent of the currently most popular representative of OO languages. The feature in Java which explicitly makes this distinction possible is the *interface concept*. In Java, an interface is a promise or a contract that is fulfilled by a class. This concept was originally found in ObjectiveC [13] and Modula-3 [12].

The most remarkable features of the Java interface concept are as follows:

1.1 Model Nature of an Interface

Interfaces are "only" a (usually not runnable) model of a class.

This means that,

- the class does not have to be fully specified at the compilation stage and
- the Java compiler also accepts this.

This is not just a model on paper nor a runnable model because the interface

- can be integrated in its entirety as a compiled entity in a hierarchy - like the class hierarchy (see Section 1.2) and
- does not need to be recompiled at a later date, as it is completed by the user of the interface (= "honorer" of the contract) in a separate class.

At the modeling stage, it is sufficient to compile the signatures of the methods offered by the interface (a signature comprises the appropriate method name and the types and names of the parameters). The philosophy behind this is that interfaces are a kind of contract which the creator of the interface offers. This contract comes into force when it is accepted by the user of the interface. The user is also referred to as the "implementer" of the interface. A clear distinction is made between modeling and implementation. The roles and tasks of the participants are also defined: The interface creator tries to provide a model interface that is as usable as possible; on the other hand, the implementer is under an obligation to actually "flesh it out". Whether the user has honored this contract is determined when the user class is compiled.

What is the origin of the term "interface"? On the one hand, it is derived from the idea of collaboration between a number of developers who keep to an agreed interface. On the other, it also corresponds to the collaboration between objects, here too the fulfillment (honoring) of certain agreements being of prime importance [3].

1.2 Single Inheritance vs. Multiple Inheritance

The interface concept is based on the principle of multiple inheritance and not just single inheritance as used within the regular Java class hierarchy.

These are concepts of object-orientation, where you can comfortably describe "is-a" relations like "An elephant is a mammal" or "A RISC processor is a processor" - elephant and RISC processor being subclasses and mammal and processor being the appropriate superclasses. Generally speaking, subclasses can inherit properties from their superclasses. By simplifying the relation, we can say: The elephant class inherits from the mammal class the property of feeding the young with milk. Or: The RISC processor class inherits from the processor class the property of having an instruction set.

By the term single inheritance the following is meant: A subclass can be in an "is-a" relation with only one superclass - eg. a motor-bike may be either a vehicle or a pet but not both. In some OO languages that allow multiple inheritance you can, however, define more complex relations, namely "is-a" relations between a subclass and more than one superclass - eg. a table may be a furniture and a burnable thing and a tradable thing. While the inheritance graph is always a tree with single inheritance, this is not the case when multiple inheritance is allowed. Much discussion has been going on in the OO community for a long time about the necessity of multiple inheritance. Some people say they cannot live without it, others consider it harmful.

Java has solved the issue in its own way:

(1) A class can only be related to a single superclass, so in the class world there is only single inheritance.

(2) An interface - consisting merely of method signatures without bodies - can be related to several superinterfaces, so in the interface world you have multiple inheritance.

(3) Across the two worlds - one class can implement several interfaces.

This will now be illustrated with a simple example. Note the different keywords interface and class.

```
interface Foodlike {
        float   cal2joule(float calories);
        float   joule2cal(float joule);
        void    decay(DecayType afterSomePeriod);
        boolean edible();
        …
}

interface Fruitlike extends Foodlike, Healthy {
    /* Philosophy: Eating is not necessarily healthy. On
    the other hand, something healthy need not have
    anything to do with eating - a walk can be healthy.
    Fruit is both edible and healthy. */
        void squish();
        …
}

class Fruit extends Food {
        …
}

interface Spherelike {
        void toss();
        void rotate();
        …
}
class Orange extends Fruit implements Fruitlike,
                Spherelike {
        ... // toss()ing may squish() me
}
```

This example shows that the classes Orange and Fruit can only inherit from a superclass Fruit or Food, while the interface Fruitlike inherits from the superinterfaces Foodlike and Healthy, and the class Orange implements the two interfaces Fruitlike and Spherelike.

The reason for this "2-class society" is the Java designers adopting the following analysis: Multiple inheritance is only highly prone to errors when complete implementations with every detail, for example attributes and method bodies, are inherited, in other words: in the *world of classes* (Orange, Fruit, Food, ...). In the *interface world*, however, only models are inherited, which cannot have the following results:

- all of a sudden, it is impossible to determine precisely where a particular detail actually "comes from" or
- certain characteristics which have been inherited from different parents are contradictory.

1.3 What about polymorphism?

However, polymorphism - another OO concept, namely the ability of a variable to refer to objects of different types - plays a subordinate role in the interface hierarchy. Polymorphism *via casting*, as used in conjunction with class hierarchies, only makes sense if two methods in one superclass and one subclass have the same signature but different bodies.

Example:

```
class Foo {
  public void f ( ) {                  // Method of the
                                       // superclass
    System.out.println ("Foo's f ( )");
                                       // Body of Foo.f()
  }
}

class Bar extends Foo {
  public void f ( ) {                  // Method of the
                                       // subclass
    System.out.println ("Bar's f ( )");
                                       // Body of Bar.f()
  }
}

public class PolymorphCasting {
  public static void main (String [ ] args) {
    Foo        myFoo        = new Foo( );
    Bar        myBar        = new Bar( );
    Foo [ ]    myFooArray   = new Foo [2];

    myFooArray[0] = myFoo;
    myFooArray[1] = myBar;

    for (int i = 0; i<myFooArray.length; ++i) {
      myFooArray [i].f ( );
    }
  }
}
```

The methods used for interfaces, however, do not have any bodies.

Also, the second type of polymorphism *via overloading* - also referred to as *ad-hoc polymorphism* - is not appropriate. It would be implemented by having several methods with the same name but with different signatures within one and the same interface. However, this is more of a hindrance for the designer of the class which implements the interface, as he/she would always be forced to implement at least two methods whether (s)he needed them or not.

Using the method cal2joule as an illustration, we will now demonstrate that it is essential for the designer of the Foodlike interface to ensure that every implementation of the interface realizes a *corresponding* method properly. About the "how", however, nothing is said at this point.

On one hand, the advantage of this *saying nothing* is that the designer does not need consider what is involved in an early stage, but on the other hand the cal2joule example below indicates a decisive disadvantage and shows what could go wrong in a case like this if the implementation cannot be checked using certain plausibility criteria.

The method cal2joule could be realized like this:

```
float cal2joule(float calories) {
        return 0.2 * calories;       // Not quite what the
                                     // designer intended !
}
```

This situation will be discussed in more detail in the next Section.

2. Critique of the Java Concept

As we have seen, although the Java interface concept is well-suited to supporting teamwork and reuse because of its ability to handle abstraction, there is still a crucial step missing as far as precision (x-axis in Fig. 1) is concerned. Occasionally, the signature of a method can leave all too much unstated. Under these circumstances, it would be important for the implementer to have a clearer idea of the expectations the interface supplier had of him/her. Also, it would often be beneficial for the supplier if (s)he could justify his/her expectations more precisely - even just to himself/herself.

For example, it would be useful to be able to state the following in the interface Foodlike:

```
float cal2joule(float calories);   "... but ensure that the
                                    conversion factor is between
                                    4.17 and 4.20."
```

The designer of the class which implements the interface can then still decide how accurately (s)he wants to specify the conversion factor: 4.2, 4.19 or 4.1868.

With this feature, it is not a question of a precondition or a postcondition but rather a constraint of the interface designer. (S)he passes it on to the designer of the class which implements the interface. We, therefore, refer to this concept as an *implementation constraint*. By this we mean a Boolean expression p in an interface S, where the class C, which implements interface S, must ensure that p is always true in C.

There is a further concept which represents a somewhat weaker requirement to be fulfilled by the implementer. We call it *knowledge requirement*. This is a requirement that the implementer (= user) must have a certain level of knowledge about the intentions of the interface designer (= supplier).

For example, a lot more could be known about the intention of the supplier, if, in addition to the methods

```
void      decay(DecayType afterSomePeriod);
boolean   edible();
```

in the interface Foodlike, there were some information relating to what the supplier thought about the context, say

```
void    decay(DecayType afterSomePeriod)
        // | {
        // |    allowedPeriod = afterSomePeriod;
        // | }
        ;


boolean  edible()
        // | {
        // |    if (new Date().getPeriod() >
        productionDate.getPeriod() + allowedPeriod)
        // |                /* new Date() = today */
        // |        return false;
        // | }
        ;
```

It can then be deliberately left open as to what is specifically meant by “period”. In the case of crispbread this might be several months, but with bottled milk only a few days. In the first case, getPeriod() should be replaced with getMonth() and in the second by getDay().

We have chosen the symbol “//|” similarly to that of VAL (VHDL Annotation Language) [1] where “--|” is used as a symbol for this kind of pseudo-comment.

This example may seem trivial, but the basic idea can be extended from food to more complex processes like autopilots or power station control.

The concept of the *type model*, which originates from Desmond D'Souza [4], seems to be a suitable solution which goes beyond informal documentation and can be automatically checked. This gives the supplier of the interface the opportunity of passing on significant information to the user about certain attributes and knowledge requirements which the supplier intends - going beyond method signatures - and to check that they have been met.

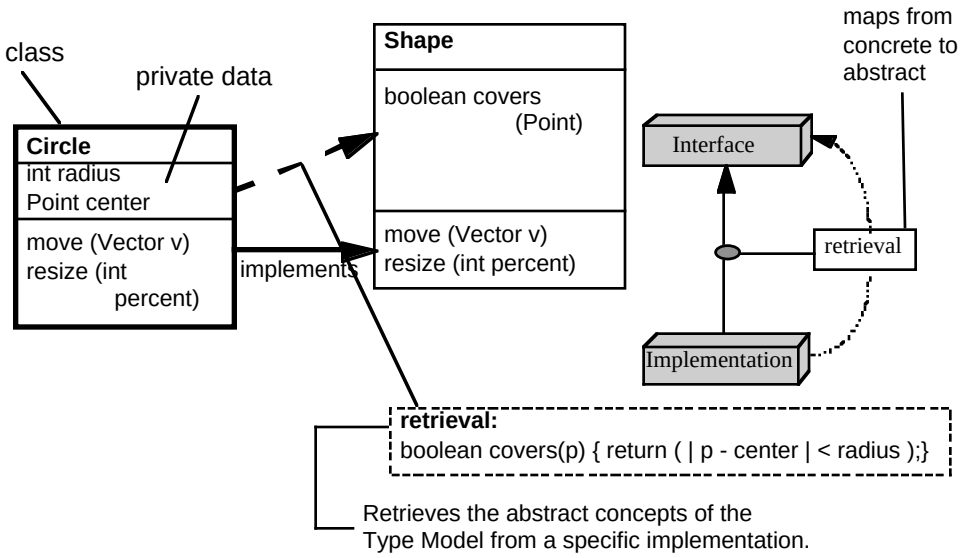

Fig. 2:
The class Circle implements the interface Shape [6]

For example, an interface Shape is to be defined in two-dimensional space and it is expected that it will be implemented by all classes which are two-dimensional geometrical figures with a certain shape (Shape). The two methods that are offered (see Fig. 2): move (= translate figure by a specified vector) and resize (= change the size of the figure by a certain percentage) are then performed by each implementation class in the way that is the most efficient for the figure in question.

With Java, however, it is not possible to pass on additional formal information beyond that offered by the signatures of the methods. In this case, for example, the supplier of the interface Shape might also want the implementer of the interface to ensure that the abstract concept

"Figure covers point" or "*Figure does not cover point"*

is implemented in a suitably specific form.

For this reason, a test mechanism referred to as *retrieval* was proposed in [5]. Apart from allowing the use of *type models* when describing interfaces (see box Boolean covers (Point) in Fig. 2), it checks whether the implementer is actually in agreement with this expectation. In our example, the user can unburden himself

of this task by introducing | p - center | < radius as a condition and so implement the corresponding abstract condition using his/her own concrete means.

The orthogonality of abstraction and precision which was mentioned in the introduction is achieved in this case. The interface designer states as abstractly as possible but with sufficient precision what his/her intentions were when (s)he was designing the interface and what (s)he expects from all implementers. This is a step towards the *shall-prototype-test principle*, which has been postulated in a general form in [11]. The knowledge requirements considered in this Section (see above) are elements of the set of "*shalls"* defined in that paper.

3. On the Way to Design Habitability

"Habitability" of program code is a concept that has been discussed for a long time by the OO-Community [9]. This is a way of referring to a new type of "mutual consideration" where the writer of a piece of code always bears in mind the users of his/her code and their knowledge and expectations:

- What must they know?
- What do they not need to know?
- How do I provide them with the relevant information?

This means that the code writer tells the user,

- what the piece of code does,
- what it does not do,
- where "traps" might occur,
- where it is relatively easy to make modifications [10],
- what effects do these modifications have.

Until tools are available to give support in this area, say in the form of pre-processors, comment lines are the only way of passing on this information.

Even in this relatively primitive form, under the most favorable conditions, the user feels "at home" in the program section, hence the term "habitability".

By "user" we do not just mean the person who calls the piece of code, but also the person who reads the listings and has to decide whether (s)he can reuse the code. The person who has written the code can also take on this role, if he or she wants to understand his or her own code, say, four weeks later.

In my opinion what has been said about making code "habitable" also applies to the work produced by designers. In this case too, it is essential to produce results in a form that the user not only understands, but also feels "at home with" to a certain extent. This feeling of well-being can be increased even further by creating a "virtual pleasant environment" in which the user feels happy because (s)he sees that the designer has been considerate enough to think about the "poor guy/girl" who might have to understand the architecture or certain design decisions [17].

4. Conclusions relating to an OO-extension of VHDL

We have made a wide curve stating some ideas which mainly stem from the area of software design. But as we already mentioned in [8], it is always wise to look to the "other side". So now let's summarize what we have learned on this curved trip.

As VHDL already supports teamwork and reuse very well at present [20], an essential goal for the designers of object-oriented extensions of VHDL should be to increase the power and scope of this support even further. The following models are available

- firstly, the Java interface concept which provides suitable approaches to abstraction and
- secondly, the type-model concept as an instrument for obtaining precision and an aid to honoring contracts.

As has been shown, the two subgoals of abstraction and precision are not mutually exclusive. They could also be very useful coordinates for an OO-extension of VHDL as

- this language already has an adequate basis mechanism in the form of the principle of separation in entity and architecture,
- there are analogies between VHDL structure descriptions and the object-oriented approach [7] and
- in the case of VAL (VHDL Annotation Language), a number of decisions have already taken it along the path towards constraint handling (assume, assert, finally, eventually, sometime) [1]. A concept of contract honoring (see above) can be constructed on these developments.

5. References

[1] L. M. Augustin et al.: *Hardware Design and Simulation in VAL/VHDL.* Kluwer Academic Publishers 1991.

[2] J. Böttger, W. Ecker: *Ada'95 - ausführbare Spezifikation in digitalen Systemen.* 3rd GI/ITG/GME-Workshop on Hardware Description Languages and Modeling Paradigms, Holzhau, Germany, Feb. 26-28, 1997, pp. 10-15.

[3] D. D'Souza: *Collaborations: Beyond subtypes.* Journal of Object-Oriented Programming, Jan. 1997, pp. 61-66.

[4] D. D'Souza: *Behavior-Driven vs. Data-Driven - A Non-Issue?* http://www.iconcomp.com/papers/data-vs-behavior/index.htm

[5] D. D'Souza: *Java: Design and modeling opportunities.* Journal of Object-Oriented Programming, Sept. 1996, pp. 14-18.

[6] D. D'Souza: *Interfaces, subtypes, and frameworks.* Journal of Object-Oriented Programming, Nov. 1996, pp. 19-22.

[7] W. Ecker: *An Object-Oriented View of Structural VHDL Description.* VIUF Spring '96, Sta. Clara, Ca., pp. 255-264.

[8] W. Ecker, M. Mrva: *Objektorientierung: Modellierungs- und Entwurfsparadigma des Jahres 2000?* 2nd GI/ITG/GME-Workshop on Hardware Description Languages and Modeling Paradigms, Darmstadt, Germany, Feb. 15-16, 1996, pp. 118-127.

[9] R. P. Gabriel: *The quality without a name.* Journal of Object-Oriented Programming, Sept. 1993, pp. 86-89.

[10] R. P. Gabriel: *Habitability and piecemeal growth.* Journal of Object-Oriented Programming, Feb. 1993, pp. 9-14.

[11] M. Heuchling, W. Ecker, M. Mrva, D. Monjau: *Das Shall-Prototyp-Test-Entwicklungsmodell.* 3rd GI/ITG/GME-Workshop on Hardware Description Languages and Modeling Paradigms, Holzhau, Germany, Feb. 26-28, 1997, pp. 122-132.

[12] J. Horning et al.: *Some Useful Modula-3 Interfaces.* Dec. 1993. ftp://ftp.digital.com/pub/DEC/SRC/research-reports/SRC-113.pdf

[13] G. Huizenga: *Objective-C.* http://www.cs.indiana.edu/classes/c304/ObjC.html

[14] J. C. McKim: *Programming by Contract: Designing for Correctness.* Journal of Object-Oriented Programming, May 1996, pp. 70-74.

[15] B. Meyer: *Object-Oriented Software Construction.* Prentice Hall 1988.

[16] M. Mrva: *Kann das Interface-Konzept von Java als Vorbild für den Vererbungsmechanismus einer objektorientierten Erweiterung von VHDL dienen?* 3rd GI/ITG/GME-Workshop on Hardware Description Languages and Modeling Paradigms, Holzhau, Germany, Feb. 26-28, 1997, pp. 85-93.

[17] M. Mrva: *Are we on the way towards design habitability?* Position Statement ECBS'97 Conf., Monterey, Ca., March 24-26, 1997.

[18] W. Nebel, W. Putzke-Röming, M. Radetzki: *Das OMI-Projekt REQUEST.* 3rd GI/ITG/GME-Workshop on Hardware Description Languages and Modeling Paradigms, Holzhau, Germany, Feb. 26-28, 1997, pp. 25-33.

[19] W. Nebel, G. Schumacher: *Konzepte objektorientierter Hardware-Modellierung.* 2nd GI/ITG/GME-Workshop on Hardware Description Languages and Modeling Paradigms, Darmstadt, Germany, Feb. 15-16, 1996, pp. 104-117.

[20] V. Preis, R. Henftling, S. März-Rössel, M. Schütz: *A Reuse Scenario for the VHDL-based Hardware Design Flow.* EURO-DAC '95, Brighton, Sept. 1995, pp. 464-469.

[21] P. van der Linden: *Just Java.* The Sunsoft Press, Java Series 1996.

[22] B. Stroustrup: *The C++ Programming Language.* Addison-Wesley 1991.

[23] R. Switzer: *Eine schöne Aussicht.* OBJEKTspektrum, Mar/Apr 1994, pp. 57-60.