European Simulation Multiconference – ESM '2001, pp. 139-145.
6-9 juin 2001, Prague, République Tchèque. Society for Computer Simulation (SCS) International.

ARCHITECTURES VALIDATION IN AN OBJECT-ORIENTED FRAMEWORK

Frédéric Mallet
Fernand Boéri
Laboratoire Informatique, Signaux et Systèmes (I3S)
UMR 6070 CNRS-UNSA
Les Algorithmes – bât. Euclide B – BP 121
06903 Sophia Antipolis Cedex France.
Frederic.Mallet@unice.fr, boeri@unice.fr

KEYWORDS
Validation, hardware architectures, object-oriented modelling, simulation, CAD.

ABSTRACT

This work presents a new method to validate hardware architecture models in an object-based modelling and simulation framework. Our incremental method called SEP consists in simulating high level models to evaluate performances of new hardware architectures relatively to critical digital signal processing applications. Efficient architecture models built early in the design process due to this high-level assessment are refined until they can be used as references helping the design of RTL models. We introduce two of the object-based mechanisms allowing reusable modelling : dynamic binding and module service. We advocate these both mechanisms were efficiently used to take complex types – such as binary decision diagrams (BDD) representing algebraic expressions – into account; and those types are used to validate architecture models, verifying the functional accuracy of high-level services relatively to the specification.

THE PROBLEM

A collaboration with VLSI Technology – a subsidiary of Philips semiconductors – leads us to design an object-oriented simulation framework to evaluate performances of digital signal processing hardware architectures very early in the design process, long time before synthesis.
In order to fulfil increasing time-to-market constraints and to adapt themselves to reducing market-life of specialised hardware architectures, chip designers need retargetable methods and tools which will fit design requirements of future architectures.
In the field of processors, tools and simulators must take into account both micro-architecture and instruction-set (see Figure 1). These tools are absolutely necessary during qualification process by potential customers, that is to say, earlier they are available, lower the design costs .
Due to its generic object-oriented model of architectures (Mallet et al. 1998), our proposed method (SEP) is able to incrementally create models with an abstraction level adapted to evaluation requirements.

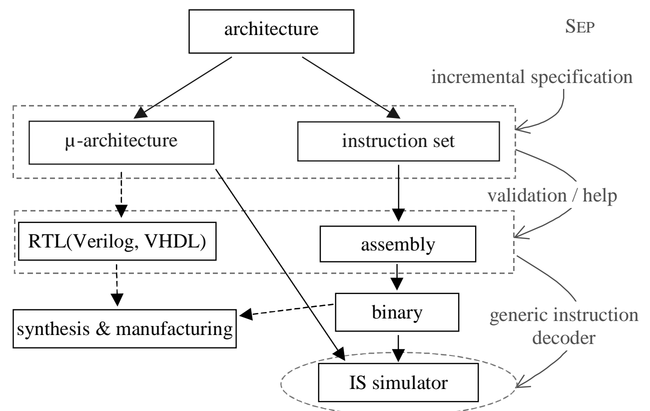

Figure 1 – Designing an architecture

Firstly, with the graphical tool, high-level models of both instruction-set and micro-architecture are created. Some critical sections of digital signal applications (GSM, TDES) are simulated and the resulting execution is analysed : number of cycles to completion and utilisation ratios of components. Secondly, the model is incrementally refined until we get satisfying evaluation criterions. Finally, the model is used as a design reference to produce RTL code in VHDL or Verilog.
Concurrent approaches (Peterson 1999; Benzakki and Djaffri 1997; Barton and Berge 1996; Sowmitri 1995), convinced by object-oriented advantages, enhance hardware description languages with object-oriented paradigms. These approaches are, according to several authors (Benzakki 2000; Mallet et al. 1998; Schumacher and Nebel 1995), difficult to use and reuse, they are very constraining in the very first steps of the design process because of their primary goals as synthesis languages.
Instead, our approach is based on the object-oriented language Java and adds upon it necessary concepts to describe hardware architectures. It looks like Jester (Antoniotti & Ferrari 1999) or JavaX (Benzakki 2000) languages. SEP however includes mains of the major characteristics of Architecture Description Languages (ADL) following criterions given by (Medvidovic and Taylor 2000) SEP is an ADL to design specialised hardware architectures.

From our generic object-oriented model of hardware architectures, we have built a modelling and simulation framework which does not depend on a specific architecture. Then, we wished we had some validation techniques in order to restrict needed tests to qualify our models.
This article introduces our method to include validation techniques into a simulation framework and advocates our model qualities which allow such an inclusion. The dynamic binding mechanism simplifies models and increases component reuse as shown by next section Then, we recall meaning and use of module services in SEP to increase model readability. Finally, we advocates the use of our dynamic binding mechanism – specific to object-oriented approaches – to verify functionality accuracy of high-level services. The whole method is illustrated with an industrial example from VLSI Technology.

DYNAMIC BINDING APPLIED TO ARITHMETICAL AND LOGICAL OPERATIONS

Due to the SEP structural model, modules can be built by gathering some components and connecting them. Each of those components throws data to other components through a connector. The receiver-component uses that data depending on its sole specification, that is modularity. Those data could either trigger a service execution or be used as parameters to complete a service. In the latter case, the service to complete may depend on types of every data parameters. Introducing polymorph operators for that kind of services is very helpful, especially to efficiently model components using some arithmetical or logical operations. Essentially components such as ALU, shifting or normalising units, and multipliers. Polymorph operators are operators which with a sole naming (e.g. addition) may represent different operations. The choice of the accurate operation is made according to operand types.
Using that mechanism, the modelling of a powerful ALU becomes very simple. Our ALU model is unique for every architectures we have modelled (RISC, DSP). It does not depend on data types, bus length or operation arity and then can be systematically used each time a component able to perform some arithmetical or logical operations has to be modelled. The graphical interface has to be specified in such a manner designers can forbid operations the architecture does not need.

This section briefly shows limitations of the basic dynamic binding mechanism of classical object-oriented languages such as Java or C++. A more detailed study can be found in (Mallet 2000). Then our mechanism is introduced. The last section presents the use of this mechanism to validate models with SEP, this constitutes the main contribution of this article.

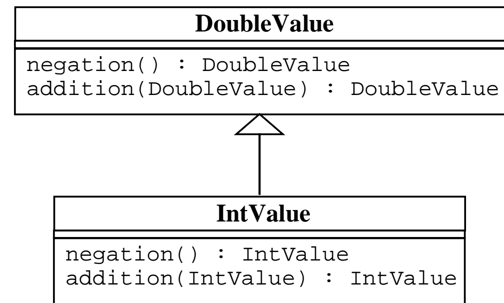

Figure 2 – Basic modelling of polymorph operators with inheritance

It is difficult to implement polymorph operators with every programming languages. And object-oriented languages allows and enforces use of polymorphism by the inheritance paradigm. This increases the lack of mechanisms to correctly implement such operators.
Let us take a simple example to illustrate that discussion. We want to define the negation and addition operations for integers – represented by the *IntValue* Class – and for real numbers – represented by the *DoubleValue* class–. A basic object-oriented modelling of these operators leads us to the definition of classes presented by the Figure 2. But, using polymorphism, we will not get the right result with the Java language. Subtype polymorphism is an ability from variables typed τ to contain references to objects which type is τ', a sub-type of $\tau : \tau' \leq \tau$.
There are two main reasons explaining this phenomenon. Firstly, it is not allowed to modify the return type of a redefined method. A method in an heir class is said to redefined a method in its parent class if and only if the method signature (method name, ordered list of parameters' types) is conserved. Thus, in the *IntValue* class, the *negation* method must return the same type as the redefined *negation* method from the *DoubleValue* class, that is to say a *DoubleValue*. The accurate method is dynamically chosen, due to the dynamic binding process, depending on dynamic type of concerned operands. Without a dynamic binding mechanism, the choice would have been done depending on the static types of references used.
The second problem comes from the *addition* method defined in the *DoubleValue* class and overloaded in the *IntValue* class. In Java, overloading is statically resolved depending on static type of effective parameters. One can say that the method $\mu 2$ is overloading $\mu 1$ if and only if $\mu 1$ and $\mu 2$ are defined in the same inheritance relation, have the same name, different signatures and eventually different return types. This induces an non symmetric behaviour between the implicit parameter upon which the dynamic binding mechanism acts, relatively to effective parameters upon which a static resolution is applied.
During invocation of an instance method, object upon which the method is applied, is implicitly present in the list of effective parameters and can be referenced as '*this*'. At runtime, everything works has if the invoked method have a supplementary parameter called '*this*' and with the type of the class declaring the invoked method. '*this*' is called the implicit parameter.

As a concrete example, the 12 (*IntValue*) + 15,5 (*DoubleValue*) operation is not treated in the same way as the 15,5 (*DoubleValue*) + 12 (*IntValue*) operation. In the first case, one invokes a method from the *IntValue* class with a DoubleValue parameter. In the second case, one invokes a method from the *DoubleValue* class with a IntValue parameter. In the latter case, the dynamic binding process will eventually select a more precise definition of the statically selected method in a class inheriting from the *DoubleValue* class (e.g. *IntValue*).

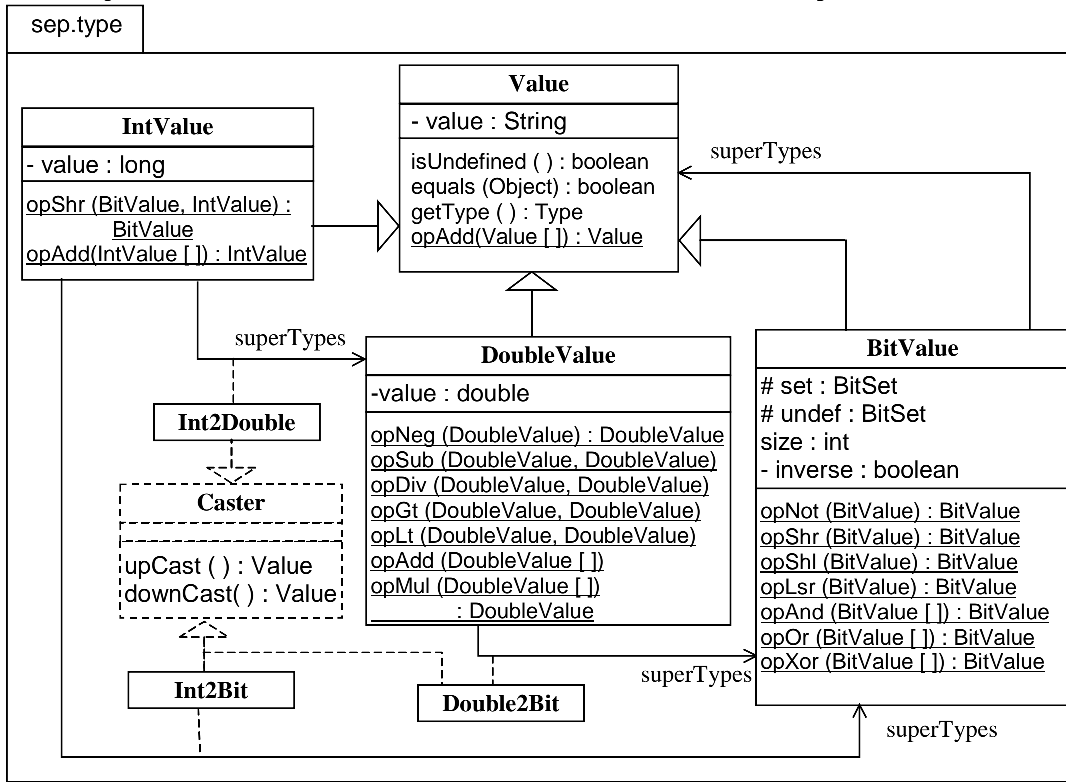

Figure 3 – The subtyping tree (Value, DoubleValue, BitValue)

SEP proposes a general, dynamic, symmetric mechanism to choose the accurate method to execute depending on operands' types. To do so, SEP must manipulate notions of type, subtyping relations and operators. This is made concrete by the definition of a meta-model for SEP types. Furthermore, extending the subtyping tree (see Figure 3) and due to this meta-model, one can define new types and new operations upon types. As an essential constraint, we have imposed ourselves to be able to add new sub types without any modifications of the super type. In (Mallet 2000), we show that several modifications are forced with the classical dynamic binding mechanism of the Java language.

To add a new type in SEP, one must define a class inheriting from the *Value* class or one of its subclasses. He defines operations as static methods, then the respective role of parameters is symmetric. Let us note as a consequence that dynamic binding is never and could not have been applied upon static methods. Furthermore, the designer has to defined super-types of the newly defined type and explicit casting operators. As showed by the Figure 3, this subtyping relation is not necessarily implemented using the inheritance mechanism. At runtime and as soon as a new type is firstly used, SEP invokes the *getSuperTypes* method to upgrade its own registry database about existing types and to build the subtyping tree. Let us note, casting operations are not transitives although the subtyping relation is. So, casting operations between *DoubleValue* and *BitValue* have to be defined, although casting operations have been defined between *DoubleValue* and *IntValue*, and between *IntValue* and *BitValue*. Transitivity may depends on coding algorithm used.

This mechanism is completely detailed in (Mallet 2000). To illustrate the extremely simple specification description we need to model a very complete ALU, let us see the Java code used by SEP:

```
import sep.type.Value;

public class Alu
        implements sep.model.ServiceProvider {
  public Value execute(Value[] values, String cop)
{
      return sep.type.Type.perform(cop, values);
  }
}
```

The *sep.type.Type.perform* method parses the subtyping tree looking for the most specific applicable method taking into account dynamic types of data carried into the *values* array. The *cop* entry is a string which represents the operation to be executed (here : add).

Then 12+15.5 or 15.5+12 operations are symmetric, in both case the *DoubleValue opAdd(DoubleValue [ ])* method is

selected, 12 is represented by an object from the *IntValue* class and 15.5 by an object from the *DoubleValue* class. Such an operation concerning both an *IntValue* and a *DoubleValue* has to be defined in the *IntValue* (the lowest common subtype between *IntValue* and *DoubleValue*). There are no applicable method is that class so we have to recursively look into lower super-types. We finally find the *DoubleValue opAdd(DoubleValue [])* method as an applicable method. 12 (*IntValue*) must be converted into 12.0 (*DoubleValue*) then, the method is invoked.
Instead, additioning two *IntValue* objects, we would have selected the *IntValue opAdd(IntValue [])* applicable method from the *IntValue* class. Java would have only taken into account static types of parameters and would have induced above-mentioned problems.

HIERARCHICAL MODELS AND MODULE SERVICES

Gathering elementary components and due to our hierarchical model, we can perform structural models called modules. The behaviour of elementary components is defined as a composition of services which can be described as Java methods or Esterel modules (Mallet 2000).
Furthermore, we can define high-level services into structural modules. These high-level services are sequential or concurrent compositions of services of elementary components constituting the module. This construction increases model readability and makes possible use of powerful mechanisms such as behaviour inheritance (Mallet 2000).

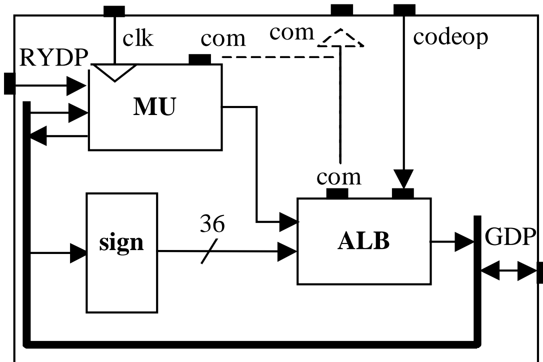

Figure 4 – Calculation unit : CU

Let us have an introduction to the specification of those services using as an example the modelling of a calculation unit extracted from a digital signal processor (DSP) core. The Figure 4 presents this calculation unit. It is mainly composed of a multiplication unit (MU) and of an arithmetical and logical block (ALB). The *sign* component is a combinatory component which performs sign extension to 36-bit for immediate incoming data. Data paths are represented with plain lines while control paths are represented with dotted lines. The inheritance UML symbol is used here to indicate that control is inherited from sub-modules MU and ALB.

Now, we want to more precisely introduce MU and ALB modules in order to advocate the benefice of using module services.
The Figure 5 shows a detailed description of multiplication unit data paths. This is an high-level structural representation of something supposed to implement some functions concerned with a multiplication purpose. It mainly aims at multiplying values contained into X and Y registers. The result has to be assigned to the P register. The P register is synchronous with the processor clock.
The first objective for module services is not to overload models with the description of control paths. The second objective is to make the description richer presenting services performed rather than a list of typed input signals which have to be triggered at the right time. Furthermore, the creation of module services makes a module description closer from an elementary component description. With those both objectives, model readability is increased.

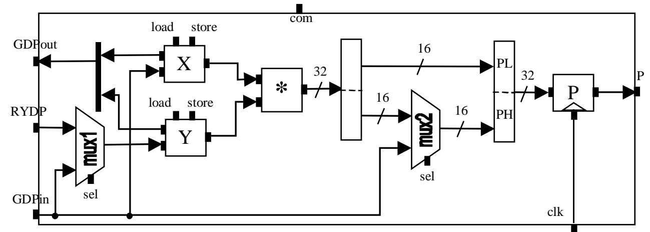

Figure 5 – Multiplication unit: MU

We have defined four basic services. The *loadX*, *storeX*, *loadY* services are inherited from the *load* and *store* services of register X, and from the *load* service of register Y. The *storeY* service is a little bit more complex. The mux1 multiplexer has to be selected in order to connect the GDP entry (port GDPin), the store service of Y register can be executed afterwards. That is a sequential composition of services. X and Y registers are then used as standard registers (read from and write to general bus GDP); the multiplier unit always performs a multiplication with values contained in those registers.
In addition, the *p := X * Y* service is defined as *mux2.sel <= 0 ; (storeY || storeX)*. It selects *mux1* and *mux2* multiplexers and loads X and Y registers.
Finally, there is also the *ph := GDP* service which assigns the highest part of the register P with the value set on the GDP bus – to obtain such a behaviour one has to correctly select the *mux2* multiplexer.

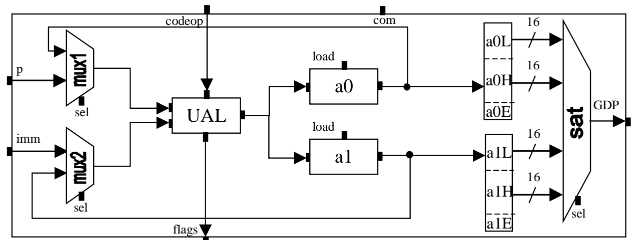

Figure 6 – Arithmetical and logical block : ALB

The Figure 6 presents data paths of the arithmetical and logical block. This module is composed of an arithmetical

and logical unit (ALU) already presented in the previous section. Results from ALU can be assigned to one the two accumulators (*load* service) a0 or a1. Operands are selected by two *mux1* and *mux2* multiplexers. In SEP, accumulators are able to memorise any data whatever its length is. In that case, they must memorise 36-bit data. They can be decomposed into three parts; the both lowest are 16-bit length (a0H, a0L, a1H, a1L) and can be sent to the GDP bus across the saturation unit (*sat* : sequential unit). The 4-bit length highest part is concatenated into a status register.
The ALB module defines the *write_a0* and *write_a1* services to write into accumulators; the *GDP := a0H*, *GDP := a0L*, *GDP := a1H*, *GDP := a1L* services to trigger the saturation unit and send a 16-bit part of one of the two accumulators to the GDP bus (a0H, a0L, a1H, a1L); and the *read_a0*, *read_a1*, *read_p*, *read_imm* services to select appropriated operands for ALU via multiplexers.
To conclude the description of the calculation unit, the both ALB and MU modules are gathered (see. Figure 4), and the *mulacc_a0*, *mulacc_a1* services are defined. They simultaneously realise, a multiplication with data loaded from GDP and RYDP busses, and an operation between the P value and either a0 or a1 accumulator, the result is accumulated into the selected accumulator. The operation to perform – used to be addition or subtraction – depends on the operation code (*codeop*) sent to the ALU.
Thus, we realised an hardware unit which is supposed to be able to realise a multiplication-accumulation operation. This operation is enough complex to want it to be validated. So, the next section presents how the use of our dynamic binding mechanism has allowed to validate that the built service realised the accurate wished function.

MODEL VALIDATION AND BINARY DECISION DIAGRAM

The Principle

We aim at validating the function accuracy for a module service. The previous section shows how complex the creation of such a service could be. Our tool automatically verify, that a module service implements a given function. To fulfil this requirement, we compare the algebraic expression really computed by the created module service with an algebraic expression of the function the designer would like to perform.
The method is very simple, but computers cannot easily manipulate algebraic expressions. This is mainly, because the canonical form is extremely complex to be obtained. So, automatic comparison between algebraic expressions is not so easy to reach and could be time-expensive. Then, we have decided to use binary decision diagrams (BDD : Binary Decision Diagram were introduced in 1978 by Akers, to represent boolean functions ($B^n \rightarrow B$) in a reduced form.). More precisely, we have chosen a form of decision diagram which easily expresses expressions we used to manipulate. BDD have an easy to obtain canonical form that makes linear, relatively to the number of nodes in diagrams, the cost of a comparison. The chosen form is called K*BMD (Kronecker Multiplicative Moment Diagram.) and has been previously introduced in (Drechsler & al 1996). This form is very efficient to represent words ($B^n \rightarrow Z$ function), to associate an integer at n bits. Elementary forms of BDD needs n diagrams to represent a n-but words, arithmetical operations on words are then very expensive.

For any module and any command we have to perform the four following steps (see Figure 7):
1. Build the K*BMD from the n-bit entry words;
2. Apply commands to be validated : call a module service or sequences of basic services. We obtain on outputs ports a K*BMD which represents the algebraic expression of the processed function ;
3. Express the function we wished we have processed and convert it into a K*BMD ;
4. Compare both expressions (resulted and wished) to deduce functional accuracy of the applied service.

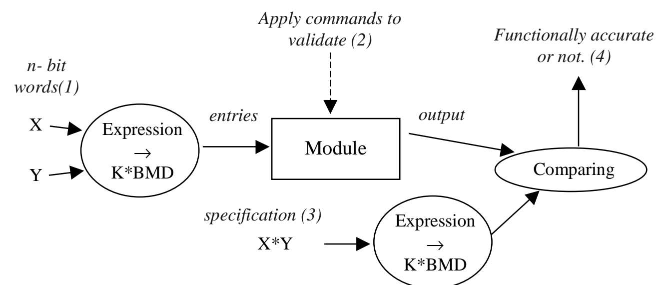

Figure 7 – Schematic of functional validation mechanism in SEP.

Using K*BMD With SEP

The major problem now is to introduce the K*BMD type into SEP framework. K*BMD are quite complexes, they are directed graphs. There are two main extensions from BDD which are based on Shannon decomposition of boolean functions. The first extension consists in mixing multiple decompositions (Shannon, positive Davio and negative Davio), this makes description more compact. The other extension consists in adding a pair-value $(a, m) \in Q^2$, on edges. If the leaf represents the function, then the valued graph represents the $a + m \times \mathbf{f}$.
The main interesting point on pair-valued graph is that the translation from a BDD represention to a K*BMD representation is as simple as selecting right values and restraining to the use of the Shannon decomposition. This equivalence is not quite so simple with others forms of BDD.
Finally, the last point is that every arithmetical and logical operations we often used for description languages can be efficiently processed with K*BMD : ALU operations, multiplication, shifting, bit-selection and bit-concatenation. The two last are in general more complexes, but some heuristics allows good time results most of the time (Höreth and Drechsler 1999).

A complete description on BDD can be found in (Bryant 1992). K*BMD are detailed in (Höreth and Drechsler 1999).

Due to our dynamic binding mechanism, we can introduce the K*BMD type in SEP, without any modification neither of the tool, nor of models. We just have to get a K*BMD implementation, to define the new type *K_BMD* and to add it into the subtyping tree (see Figure 8). The *K_BMD* class inherits from the Value class, SEP components will be able to send and receive data of *K_BMD* types.

K_BMD is defined as a subtype of the *IntValue* type because every operation applicable on integers can be applied on K_BMD. Let us remind that K_BMD are function from $B^n \rightarrow Z$. Basic operations on words are defined to operate on K_BMD.

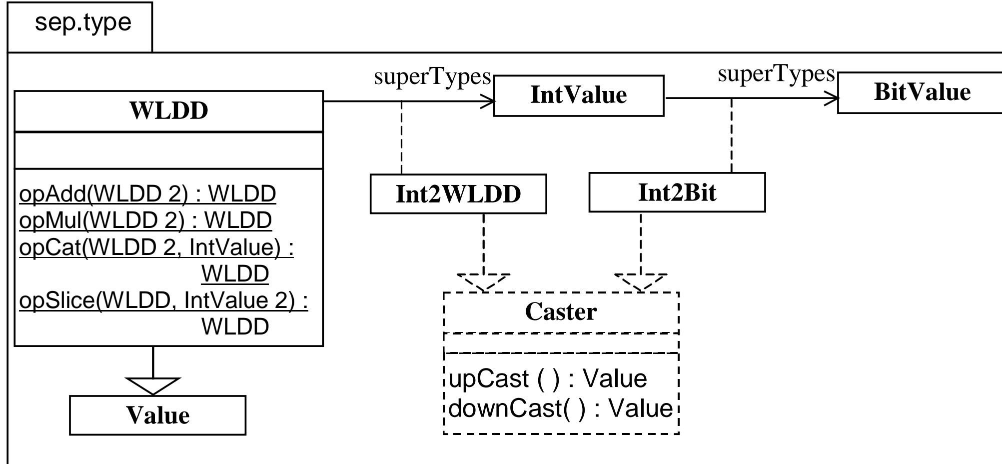

Figure 8 – Defing K_BMD type.

Then, K*BMD operations will be called when a module manipulates K*BMD data, and normal operations will be used when the same module manipulates *IntValue*. Just the simulation entries has to be modified. We can observe the manual application of this mechanism on a simple example.

An Example

Let us take the calculation unit presented in previous sections. We would like to validate the **codeop=Add ; mulacc_a0 ; clk** service where **clk** is the synchronous loading of p register with the system clock. We have to assert that when those services are sequentially applied, the accurate multiplication-accumulation behaviour is processed with two clock cycles.

To reach that objective, the CU module is instantiated into the SEP simulation framework. The accumulator a0 is set with a K*BMD representing a 36-bit word : $A0 = \sum_{i=0}^{35} \left(a_i \times 2^i \right)$. Both RYDP and GDP_{in} inputs are initialised with $RYDP = \sum_{i=0}^{35} \left(rydp_i \times 2^i \right)$ and $GDP_{in} = \sum_{i=0}^{35} \left(gdp_i \times 2^i \right)$ K*BMD. Then, the **codeop=Add ; mulacc_a0** service is invoked on th CU module. The p register is triggered to simulate the clock cycle (**clk**). The resulting K*BMD, obtained as an input for the accumulator, is compared to the K*BMD obtained from the $A0+RYDP \times GDP_{in}$, expression. Those both K*BMD match to confirm the accuracy of the service.

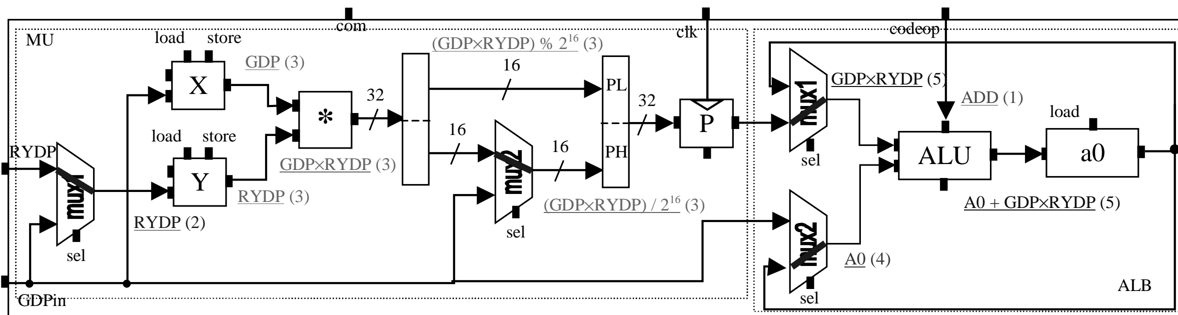

Figure 9 – A validation scenario.

To be convinced, we can manually look at the evolution of the computation (see Figure 9) at each instant.

First instant : the code operation *Add* is set as an input for the ALU.
Second instant : *MU.mux1.sel<=0 || MU.mux2.sel <= 0 || ALB.mux1.sel=2 || ALB.mux2.sel=0.* Multiplexers are selected.
Third instant : *MU.Y.com=store || MU.X.com=store*. Both X and Y registers are loaded and output expression are multiplied. The multiplication result is decomposed into the highest part *(GDP*RYDP)/16* and the lowest part *(GDP*RYDP)%16*. Then both parts are concatenated back.
Fourth instant : *ALB.a0.load.* The a0 is activated, the *a0* value is transmitted towards the second ALU input.
Fifth instant : the p register is activated, the $GDP \times RYDP$ expression is transmitted toward the first ALU input. The calculated result is then $A0 + GDP \times RYDP$.

As an example, we manually obtained the result reasoning on algebraic expressions. SEP can obtain it in simulation using K*BMD. Due to the dynamic binding mechanism, such a process can been introduced with no additional cost except the implementation of a library to manipulation K*BMD. One just has to include a new type into the subtyping tree.

CONCLUSION

We have presented a technique based upon use of decision diagrams to validate hardware architectures models. We aim at building high-level, expressive, reusable models to describe hardware architectures. The proposed mechanism are coming from the field of object-oriented modelling and allows a deep modelling of complex types such as decision diagrams. In this case, BDD are used to validate high-level object-based models.
This general method may be reused in any simulation framework where designers want add some validation features. The dynamic binding mechanism presented may be used to take into account other types depending on needs. We will continue our modelling effort and try to apply our method to some higher-level applications in the field of the system-level modelling of multi-core chips.

REFERENCES

Antoniotti, M. and Ferrari, A. 1999 "Jester : a Reactive Java extension proposal by Esterel Hosting."
http://www.parades.rm.cnr.it/projects/jester/jester.html.

Barton, D. L. and Berge, J-M. 1996 "A proposed Design Objectives Document for Object-Oriented VHDL."
The RASSP Digest - Vol. 3, september 1996.
http://rassp.aticorp.org/newsletter/html/96sep/news_18.html

Benzakki, J. and Djaffri, B. 1997 "Object-Oriented Extensions to CHDL : The LaMI Proposal"
IFIP 1997. Chapman & Hall, 334-347.

Benzakki, J. 2000 "Objets pour la modélisation de Systèmes Matériels : intérêts, évolutions et tendance."
Habilitation à diriger des recherches, d'Evry Val d'Essonne University, january 2000.

Bryant, R.E 1992 *"Symbolic boolean manipulation with ordered binary-decision diagrams"*
ACM Computing surveys, 24(3) :293-378, 1992.

Drechsler, R; Becker, R. and Ruppertz, S. 1996 "K*BMDs : A new Data Structure for Verification"
IEEE European Design & Test Conference (ED&TC'96), Paris 1996, 2-8.

Höreth, S. and Drechsler, R. 1999 "Formal Verification of Word-Level Specifications"
IEEE Design, Automation and Test in Conference (DATE'99), Munich, 1999.

Mallet, F. 2000. "Modélisation et Evaluation de Performances d'architectures matérielles numériques."
PhD thesis, Nice-Sophia Antipolis University, december 2000.

Mallet, F.; Boéri, F. and Duboc, J-F.. 1998 "Hardware Architecture Modelling using an Object-oriented Method."
24th Euromicro conference, september 1998, vol I, 147-153.

Medvidovic, N. and Taylor, R.N. 2000 "A classification and comparison Framework for Software Architecture Description Languages.",
IEEE Transactions on Software Engineering, Vol.26, No. 1, january 2000.

Peterson, G.D. 1999 « Proposed Language Requirements for Object-Oriented Extensions to VHDL. »
Proceedings of Forum on Design Languages, FDL'99, France, september 99.

Schumacher, G. and Nebel, W. 1995 "Inheritance Concept for Signals in Object-Oriented Extensions to VHDL."
EURO-DAC'95 with EURO-CHDL'95.

Sowmitri, S. 1995 "Object-Oriented VHDL Provides New Modeling and Reuse Techniques for RASSP."
Vista RASSP Program Manager. The RASSP Digest - Vol. 2, No. 1, 1st. Qtr. 1995
http://rassp.aticorp.org/newsletter/html/95q1/news_6.html