Using Axiomatic Specifications for Hardware System Design

Marc Aiguier, Stefan Beroff, Laurent Freund, Gilles Bernot and Michel Israël
La.M.I.
Laboratoire de **M**athématiques et d'**I**nformatique
Université d'Evry Val d'Essonne
Bd des Coquibus
F-91025 Evry Cedex, France
e-mail: {aiguier,beroff,freund,bernot,israel}@lami.univ-evry.fr

1 Introduction

As complexity of Hardware/Software systems increases, it becomes impossible to manage this complexity without formal methods. Formal specification allows to rigorously prove that the final implementation behaves in the manner specified. Today, computer systems involve a strong imbrication between Hardware and Software parts. For example, a processor is not only a hardware resource, software is a plain part of it, and we have to be able to do a well partitioning of hardware and software parts. For some applications (e.g. telecommunications), it is important not to induce any constraint on the Hardware/Software partitioning at the specification level. Consequently, it is of first interest to use an homogeneous specification language for the system as a whole. Formal approaches have already been studied, particularly the specification of the IEEE float using the language Z [Spi92] [Jon90]. Approaches such as Z or VDM are "model oriented" languages. We decided to use a "property oriented" language, algebraic specification with exception handling [BGA94] , which allows to reach a very abstract specification style where no hypotheses about the future implementation are made.
Algebraic specifications allow: checking **coherence** and **completeness** of a specification, **proof**: before and after refinement, **stepwise enrichment**: specify and prove step by step all along the development process, successive **refinements** up to **code generation**. The usefulness of algebraic specifications is widely recognized for Software, the main purpose of the work reported here was to check if algebraic specifications are also able to specify Hardware.
We focused on the specification of the RISC architecture. For that purpose, we decided to choose the DLX processor as a case study, because it is a representative example of the RISC technology (see [HP90] and [HP94]). Our approach was to first write a full abstract specification based on well established mathematical abstractions of the real-life data structures (integers, natural numbers, ...) and the observational behavior of the execution unit. Then we wrote a low level specification where DLX is described by its lower level data structures (array of

Booleans). Lastly, we performed the most relevant proofs of the correctness of this abstract implementation. Proofs are possible at all levels of specifications and during the abstract implementation.

2 The advantages of rigorous formal methods

2.1 The role of formal specifications

One of principal targets of all methods and technics of hardware / software system design is to obtain *flexible* (i.e. one can easily have a system advanced) *reusable* (i.e. either for simpler purposes or like a basis of some larger system) and *correct* systems (i.e. the system does what it is supposed to do). Formal specifications are good means to reach these three properties for two reasons:

- Formal specifications are written according to rigorously and mathematically established *syntax* (i.e. what a specifier is allowed to write) and *semantics* (i.e. what is a correct system behavior with respect to the specification). Consequently, they never contain ambiguities. So, the understanding of the specification by the reader and the verification of its correctness are better achieved. Then, the mistakes of specifications fraught with consequences are avoided.

- Formal specifications allow to rigorously write specifications in describing "*what the system is supposed to do*" without to be interested by "*how it is done*". It is true that it is not easy to make the most abstract specification even if well known mathematical structures facilitated this issue. Indeed, it is a strong discipline, a kind of "pedagogical effort." However, this effort is profitable, especially if the specified component is used (or reused) several times: the abstraction effort to understand the purpose of the component has not to be done by the readers. Moreover, resulting specifications are clearer, terser and more legible. Consequently, the use, the reuse and the maintenance of system are facilitated.

Moreover, formal specifications, to be fully usable, should offer a set of rules (also called *calculus*) allowing to prove if certain properties are ensured by a given specification. The set of rules has to be *sound* (i.e. all properties proved from these rules must semantically be true) and may be *complete* (i.e. all properties semantically true can be proved from the set of rules). The interest of such a calculus is to be able to *directly* and *syntactically* reason on a given specification. Consequently, in practice, a specifier is not obliged to understand all the "so-complicated mathematical considerations" involved by the semantics. It is only sufficient that the specifier has an intuitive idea of what his or her specification

means, provided that the specifier can check the required properties by using the calculus.
Lastly, as a specifier can write very abstract specification, most of specifications that he or she manipulates, are often not executable. It is not always possible to directly obtain a program which verifies the specification or a prototype of the specification (e.g. by a rewriting system obtained from the specification by using the Knuth-Bendix algorithm [KB70]). Consequently, it may be necessary to perform refinement steps before to obtain such a prototype (see Figure 2.1). In the formal specification frameworks, these steps of refinement are often called *abstract implementation*. The aim of abstract implementations is to introduce at each step of refinement new design decisions. These decisions may be the choice of a special algorithm, or the implementation of some abstract operations by lower level ones, or the implementation of some data types (e.g. the implementation of stacks by an array and a natural number). The formal approaches give means to prove the implementation correctness for each refinement step.

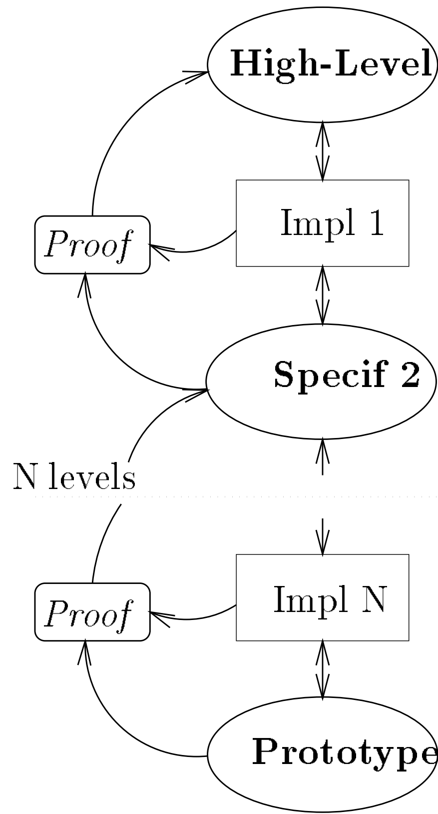


Figure 2.1

All these steps can be aided by graphic editors, formula editors, libraries of already implemented specifications, browsers, etc.

2.2 General algebraic specification framework

We may distinguish two classes of formal specification frameworks: the first one is called "model oriented" and the second one is called "property oriented" or "axiomatic".

In the model oriented approaches (VDM [Jon90], Z [Spi92], B [Abr94],...) we build an *unique* model from a lot of built-in data structures and construction primitives that the specification language offers (thus, model oriented approaches more or less induce several choices of implementation from the beginning of the specification stage). Consequently, a program is correct with respect to its specification if it has the "*same behavior*" that the model specified.

In the property oriented approaches, the specifier gives a set of "functionality names" (also called a *signature*) and a set of properties on these functionalities (often called *axioms*). These properties describe the abstract behavior of the functionalities. Consequently, the semantics associated to a specification in a declarative style is defined by a set of models which is a subset of models that one can build on the signature. Each one represents a possible implementation of functionalities of the system under specification. Then, a program is correct with respect to the specification if the model associated to this program is a model of the specification under consideration. Also, a specification is *consistent* if there exists at least one model which satisfies this specification.
So, unlike model oriented approaches, specifications in an axiomatic style are allowed to be incomplete, without loosing a rigorous semantics.

Among property oriented formal specifications, the most popular approaches are based on algebraic semantics (also called *algebraic specifications*) ([GTW78], [EM85]). The paradigm of algebraic specifications is rather a way of thinking semantics, with a common set of mathematical tools (based on category theory) to establish basic properties of the proposed specification formalism, and with a syntax mainly based on equalities. The main advantages of algebraic specifications is that it is a good mean to deal with proof, stepwise refinement, typing issues, exception handling, test, modularity, object oriented, reusability, etc.
This approach is new in the hardware design, but already experimented in the software domain (e.g. algebraic specification of embedded software of a subway [DG94], algebraic specification of a Pascal compiler [DS83], algebraic specification of a subset of the UNIX file system [BGM89] or on the data base of the environment specifications ASSPEGIQUE [Cap87], telephone switching [BH85], ...).

3 Introduction to axiomatic specifications

3.1 Algebraic specifications without exception handling

Succinctly, an algebraic specification $Spec$ is defined by a signature Σ and a set of axioms Ax.

A *signature* Σ is a set of type names S (often called sorts) with a set of function symbols F, each one provided with a profile of the form: $s_1 \times \ldots \times s_n \rightarrow s$ where $s_i \in S$ and $s \in S$ (n may be equal to 0). For example, the booleans signature is defined by the set of sorts: $S = \{BOOL\}$ and the set of operations: $F = \{true, false : BOOL,\ or, and : BOOL \times BOOL \rightarrow BOOL, ...\}$.

An algebraic model A on a signature Σ, also called Σ-algebras, is defined by:

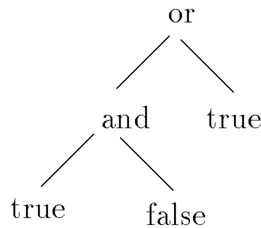
- a family of sets of values indexed by S, also denoted by A, (i.e. $A = \{A_s\}_{s \in S}$). Each set A_s is called the *carrier* of the sort s. For example, we may associate the following carrier for the sort $BOOL$: $A_{BOOL} = \{0, 1\}$.
- a set of functions $f^A : A_{s_1} \times \ldots \times A_{s_n} \rightarrow A_s$, one for each operation $f : s_1 \times \ldots \times s_n \rightarrow s$ belonging to F. For example, we associate the usual semantics to boolean operations: $true^A = 1$, $false^A = 0$,

x	y	$x\ and^A\ y$
0	0	0
0	1	0
1	0	0
1	1	1

and

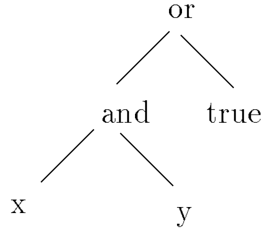
x	y	$x\ or^A\ y$
0	0	0
0	1	1
1	0	1
1	1	1

From all signature Σ, we can build a special algebraic model: the model of ground terms, denoted by T_Σ, where the family of carriers contains all terms inductively defined from the functionalities in Σ: a term of T_Σ can be represented by a tree where leaves are constants (operations "c" with a profile of the form: $c : s$) of the signature. For example, the ground term "($true\ and\ false$) $or\ true$" is represented by:



Let Σ be a signature over a set of sorts S. Let X be a family of sets indexed by S (i.e. $X = \{X_s\}_{s \in S}$). For each s belonging to S, X_s contains *variables* of type

s. We denote $T_\Sigma(X)$, the model of terms built on the signature Σ by considering the variables belonging to X_s as additional constants of the type s. We call an element belonging to $T_\Sigma(X)$, a term with variables in X: a term of $T_\Sigma(X)$ can be represented by a tree where leaves are either constants of the signature or variables. For example, the term with variables "$(x\ and\ y)\ or\ true$" is represented by:



An *axiom* is a well-formed formula inductively builds from equations "$t = u$ "where t and u are terms with variables of the same type, connectives in $\{\wedge, \vee, \neg, \Rightarrow\}$ and quantifiers in $\{\forall, \exists\}$ (free variables are supposed universally quantified). For example, we may want the following axiom for booleans structure:

$$x\ and\ false = false$$

An *algebraic model satisfies* an equation "$t = u$" if for every assignment $\sigma : T_\Sigma(X) \rightarrow A$, we have: $\sigma(t) = \sigma(u)$. The satisfaction of the formulas is inductively defined from the satisfaction of equations and the usual truth tables associated to connectives and quantifiers.

The semantics of a specification $Spec = (\Sigma, Ax)$ is represented by the class of Σ-algebras which satisfy each axiom of Ax.

Example 3.1: Let us give the complete arrays specification (it will be used later on):

$S = \{ARRAY, RANG, ELEM\}$				$/*\ Names\ of\ sorts\ */$
$F =$				
$create :$		$\longrightarrow$	$ARRAY$	$/*\ initial\ array\ */$
$store___ :$	$ARRAY\ RANG\ ELEM$	$\longrightarrow$	$ARRAY$	$/*\ assignement\ */$
$_[_] :$	$ARRAY\ RANG$	$\longrightarrow$	$ELEM$	$/*\ access\ */$

The sort $RANG$ is the sort of array positions and the sort $Elem$, the sort of array elements.

By convention, we will denote by t the variable of sort $Array$, i and j those of the sort $Rang$, and x and y those of the sort $Elem$.

Ax =
$create = store(create, i, e)$
$i = j \Longrightarrow store(store(t, i, x), j, y) \; = \; store(t, j, y)$
$i \neq j \Longrightarrow store(store(t, i, x), j, y) \; = \; store(store(t, j, y), i, x)$
$store(t, i, x)[i] = x$

The first axiom means that the empty array contains the constant e everywhere. The second and third axioms mean that the assignment in an array is commutative except for equal positions. In this case, the last assignment removes the first one. Lastly, the last one means that taking an element at a given position gives the last element put at this position.

3.2 Algebraic specifications with exception handling

From the general algebraic framework, a lot of formalism have been developed (unified algebras [Mos89], typed algebras [MSS89], order sorted algebras [Gog78], partial algebras [BW82]...).

In this article, we choose to use the formalism of algebraic specifications with exception handling [BGA94]. It is suitable to specify the DLX processor which has constraints on the data representation (e.g. we consider some data represented on 32 bits). This formalism is able to treat all the exceptional cases and exception handling features, including the following ones: implicit propagation of exceptions, "intrinsic" exceptions which are related to the underlying data structure (for instance, popping an empty stack or applying the predecessor operation on zero for natural numbers), exceptions which are relied on "dynamic" properties (as an access to a non-initialized array cell), exceptions which are due to certain limitations (mainly bounded data structures), recovery (as the exception handler that recovers every overflow value on maxint).

The main idea underlying to the theory of exception algebras is that two terms can result on the same value whilst being differently labeled.

An *exception signature* ΣL is an usual signature Σ (see Section 3.1) to which we add a set of labels L and a special label Ok. The label Ok characterizes the normal cases. Exception names and error messages are reflected by the labels belonging to L.

An *exception algebra* $\mathcal{A}$ on an exception signature (Σ, L) is a Σ-algebra A provided with, a set of terms, denoted l_A ($t \in l_A$ means that the term is *labeled* by l).

An *axiom* is inductively defined as previously but we add labeling atoms of the form "$w \, \epsilon \, l$" where w is a term with variables (i.e. an element of $T_\Sigma(X)$) and l is a label belonging to L. Such an atom means that w *is labeled by the exception* l (see example below).

As consequence of this approach, (labeled) terms must also be considered as "first citizen objects." An exception algebra $\mathcal{A}$ *satisfies* a labeling atom "$w \, \epsilon \, l$" if for

every assignment $\sigma : T_\Sigma(X) \rightarrow T_\Sigma$, we have: $\sigma(w) \in l_A$. Intuitively, $\sigma(w) \in l_A$ with $l \neq Ok$ means that the calculation defined by $\sigma(w)$ leads to the exception name l. The satisfaction of general formulas is defined from the satisfaction of atoms as above (see Section 3.1) .

An *exception specification* is a quadruple $(\Sigma, L, GenAx, DefAx)$ where $GenAx$ is a set of axioms, called "generalized axioms", and $DefAx$ a set of axioms, called "default axioms". The axioms of an exception specification are separated in two parts in order to preserve clarity and terness.

- $GenAx$ is devoted to exception handling. The purpose of $GenAx$ is to specify the exceptional cases: when to raise exceptions and how to recover them.
- $DefAx$ is entirely devoted to the normal cases (when no exception has been raised). These axioms are called default axioms because terms under consideration in this part are by default labeled by Ok and $\sigma(w) \in Ok_A$ means that the calculation defined by $\sigma(w)$ is a "normal" calculation (i.e. it does not need an exceptional treatment).

Example 3.2:

Let us consider the bounded natural number specification with the successor and sum operations. Let us assume that every calculation of the form $succ^i(0)$ $(i \geq Maxint)$ is labeled by $Overflow$. Lastly, let us consider an exception handler that recovers every $Overflow$-value on $Maxint$. We obtain the following exception signature:

$S = \{NATB\}$ $/*$ *Name of the sort* $*/$				
$F =$				
$0 :$		$\longrightarrow$	$NATB$	
$succ\ _; :$	$NATB$	$\longrightarrow$	$NATB$	$/*$ *successor* $*/$
$_ + _ :$	$NATB\ NATB$	$\longrightarrow$	$NATB$	$/*$ *sum* $*/$
$L = \{Overflow\}$				

and following specification:

GenAx:

$succ^{Maxint}(0)\ \epsilon\ Overflow$

$succ(n)\ \epsilon\ Overflow \Longrightarrow succ(n) = n$

$(succ(n)\ +\ m)\ \epsilon\ Overflow \Longrightarrow (n\ +\ succ(m))\ \epsilon\ Overflow$

$n\ +\ m = m\ +\ n$

DefAx:

$n\ +\ 0 = n$

$n\ +\ succ(m) = succ(n\ +\ m)$

The first axiom of $GenAx$ specifies the $Overflow$ domain. The second axiom specifies the exception handler which consists to recover every Overflow-value into $Maxint$-value. Lastly, the last two axioms describe general properties on every terms (i.e. both terms labeled by Ok or $Overflow$).

$DefAx$ specifies the usual operation "+" in all normal cases. As we have already seen, every axiom belonging to $DefAx$ is implicitly labeled by Ok. Consequently, in both $DefAx$-axioms above, we implicitly have: $n \epsilon Ok$, $m \epsilon Ok$, $n + succ(m) \epsilon Ok$ and $succ(n + m) \epsilon Ok$.

According to the specification defined above, we can consider the exception algebra $\mathcal{A} = (A, \{l_A\}_{l \in L \cup \{Ok\}})$ defined by: $A = [0 \ldots Maxint]$.
The operations $0^{\mathcal{A}}$, $succ^{\mathcal{A}}$ and $+^{\mathcal{A}}$ are defined as usual on integers with the following restriction: $succ^{\mathcal{A}}(Maxint) = Maxint$.

4 An informal specification of the DLX

DLX has been chosen because of its architecture. It is characteristic of all the RISC architectures. Before formally specifying DLX, let us consider a DLX's informal description.
"*What informations do we need to informally specify DLX ?*" As every architectural description, relevant informations are:

- the instruction set
- data used, their format and their addressing modes
- exceptions
- general resources

In the four next sections, we informally describe the DLX's behavior (see [HP90] and [HP94] for more complete description).

4.1 The instructions set

As usual, the instruction set contains two kinds of information:

- **The operations**
  - memory access (Load, Store)
  - ALU operations (integer and floating point)
  - Control operations (branches and jump)
- **The instruction format**

- register-register *(destination, operand1, operand2)*
- register-instructions and short branch *(destination, source, immediate)*
- Branches and jump *(immediate)*

In a pedagogical purpose, we only consider a subset of the whole instruction set in this article (see [HP90] and [HP94] for more details).

4.2 The data format and addressing modes

To correctly define registers and the memory (and also the whole set of buses), we have to describe all data types manipulated by DLX:

- Integer data type is encoded by either word of 8, 16 or 32 bits.
- Floating-point data type is encoded by either word of 32 bits for simple precision or word of 64 bits for double precision.

There is only one addressing mode, defined by:

Address = Register + offset

4.3 The exceptions

Exceptions occur in the following cases:

- Hardware or software Interruptions
- ALU exceptions
- Instruction errors
- ...

4.4 The general resources

At this description level, we only define the registers files: GPR and FPR. Other specific registers (e.g. the program counter) can be deduced from the instruction set.

In order to simplify the specification process, the following resources are used:

GPR	32 register to store integers
FPR	32 register for floating-point numbers.
R0	1 register coded to the 0 value.
SR	1 status register.
IR	1 instruction register.
MDR	1 memory data register.
MAR	1 memory address register.
IAR	1 interruption register.
IAR	1 interruption register.
PC	1 program counter.
TVR	1 TRAP value register.

5 The DLX algebraic exception specification

From the DLX's informal description, we formally specify it, using the exception specification framework.
In a pedagogical purpose, we only specify a part of the actual DLX's behavior described above.
As we have seen, the interest of axiomatic specifications is to formally describe the most abstract behavior of a system. Thus, we have to determine the fundamental DLX properties. Let us notice that the DLX's informal description above, is composed of architectural data (*e.g. the pipeline*), external characteristics (*e.g. the memory is 32-bits addressable*) and functionalities (*e.g. integers operations*). From the user point of view, the fundamental DLX properties only concern external characteristics and functionalities. The reason why architectural data do not belong to these properties is that they describe the "how part" and not the "what part" of DLX. Consequently, to completely specify the DLX behavior, we have to consider two levels of specification:

- a high level specification where concrete implementation is not considered.
- a low level specification for which some implementation choices are taken into account.

The mean to link both specifications is defined by *abstract implementation theories* (see Section 5.2.1).

The approach followed is to write a fully abstract specification based on the abstraction of the real-life data structures (integers, natural numbers...) and the observational behavior of the DLX processor (Section 5.1). Then, we give a low level specification where the DLX processor is described by its lower level data structures (typically arrays of Booleans) (see Section 5.2). This is indeed one step of implementation.

5.1 The high level specification

The high level specification will only express what DLX users need: the DLX's instruction set and the linking of instructions. Consequently, we need to specify both to reach our purpose [1].

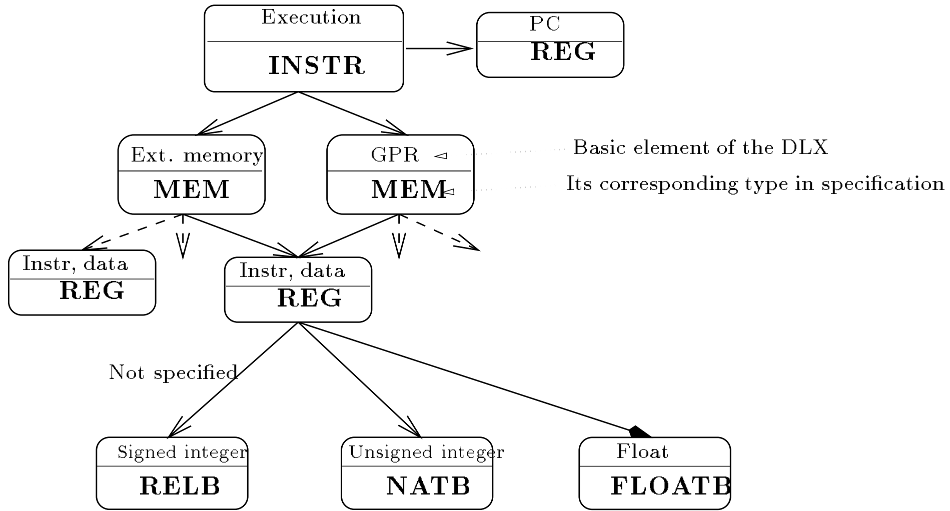
At this level of description, well known abstract data types provide the basis of the specifications. They are typically used to describe user-defined data types created to represent some abstraction of a real-life entities. The high level specification may seem to be very abstract regardless to what we want to specify. However, we should not forget that an abstract specification specifies a set of requirements to describe the system properties. For the sake of clarity, relations between abstract data types and DLX entities are given in Figure 5.1. For each entity of the DLX processor, a corresponding abstract data type specifies the behavior of the operations existing in real-life. For example, a unsigned register is a bounded structure (due to the finite representation used to memorize its value). The operations which can be performed on such a structure are sum, subtraction, comparison, and so on. So, it is obvious that the bounded natural number data type represents a good abstraction of unsigned integers. In fact, this one just keeps the interesting properties of unsigned registers (see the bounded natural numbers specification in Example 3.2). The high level DLX specification is composed of 5 modules:

- **RELB, NATB**
  Signed and unsigned integers.
- **FLOAT**
  floating-point numbers.
- **REG**
  Memorization element.
- **MEM**
  A list of memorization elements: extern memory and register file.
- **INSTR**
  Instruction scheduling.

Each one is representative of the classical specifications which can be found when trying to specify Hardware.

Schematically, DLX is represented by the following figure.

[1] In fact, to keep our pedagogical way, we only describe a subset of the complete instruction set. This subset has been chosen to represent all different types of instructions.



Graph of the specification

Figure 5.1

We do not specify the signed integers data type because it is similar to unsigned integers specification with respect to the operation predecessor (*pred*) and the label $Underflow$.

5.1.1 Specification of the float registers: FLOAT

Classically, a float register is defined by: a sign, a mantissa and an exponent. We represent signs by booleans , mantissas by natural numbers, and exponents by integers (BOOL, NATB, RELB specifications). Consequently, it is unnecessary to specify new data types. Likewise, we do not need to specify new operations. For example, a sum of two exponents is defined by the integer sum. Thus, we only focus on the specification of float registers:

$S = \{FLOAT\}$ /* *Name of the sort (type)* */				
$F =$ /* *beginning of the signature* */				
$float___$:	$BOOL \times RELB \times NATB$	$\longrightarrow$	$FLOAT$	/* *float format* */
$infinite_$:	$FLOAT$	$\longrightarrow$	$BOOL$	/* *test if value is infinite* */
$zero_$:	$FLOAT$	$\longrightarrow$	$BOOL$	/* *test if value is nul* */
$L = \{OverFloat, UnderFloat, NaN$/* *errors, Not A Number* */				

- The generator "float" means that a float register is defined by the concatenation of a Boolean (i.e. sort BOOL), a bounded integer (i.e. sort RELB), and a bounded natural numbers (sort NATB).

- The set of operations "infinite" and "zero" are observators on the float register.

The second part of the specification is defined by the following set of axioms.

GenAx:

$$\left.\begin{array}{l} eq(fract, 0) = false \\ eq(exp, EMax) = true \end{array}\right\}$$
$$\Longrightarrow float(sign, exp, fract) \ \epsilon \ NaN \qquad /*\ Not\ a\ Number\ */$$

$$Emax < exp = true \Longrightarrow float(sign, exp, fract) \ \epsilon \ OverFloat$$
$$exp < Emin = true \Longrightarrow float(sign, exp, fract) \ \epsilon \ UnderFloat$$

DefAx:

$$Emax = succ^{EMax}(0)$$
$$Emin = pred^{EMax-1}(0)$$
$$Fmax = succ^{FMax}(0)$$
$$\left.\begin{array}{l} eq(fract, 0) = true \\ eq(exp, EMax) = true \end{array}\right\}$$
$$\Longrightarrow infinite(float(sign, exp, fract)) = true$$

$$\left.\begin{array}{l} eq(fract, 0) = true \\ eq(exp, EMin) = true \end{array}\right\}$$
$$\Longrightarrow zero(sign, exp, fract)) = true$$

The GenAx-axioms are devoted to describe the domain of each label (i.e. exception name) of the signature (i.e. terms labeled by NaN, $Overfloat$ or $UnderFloat$).

Concerning DefAx-axioms, the operation "infinite" takes a float register and checks whether its value is infinite or not regardless of the IEEE 754 norm.

5.1.2 The register specification

By analyzing the instruction set, let us notice that DLX's registers contain either instructions or, signed and unsigned integers, or floating-point numbers. Moreover, instructions are encoded into three different formats:

32 … 0

NATB	NATB	NATB	NATB	NATB	REG
I / F	OPCODE	RS	RT	RD	: Register

NATB	NATB	NATB	RELB_16, NATB_16	REG
OPCODE	RS	RT / RD	IMD 16 bits	: Register

NATB	RELB_26, NATB_16	REG
OPCODE	IMD 26 bits	: Register

Instruction formats with their corresponding specification type.

Notation: The I / F stands for the integer or floating-point instruction.

Every information above are respected in the following register exception specification:

$S = \{REG\}$ /* *Name of the sort (type)* */

$F =$ /* *begin of the signature* */

$$
\begin{array}{rrcll}
instr_____ : & NATB \times NATB \times NATB \times NATB \times NATB & \longrightarrow & REG & /*\ Reg\text{-}Reg\ instruction\ format\ */ \\
instr____ : & NATB \times NATB \times NATB \times RELB_{16} & \longrightarrow & REG & /*\ Reg,\ branches,\ 16\ bits\ imd\ */ \\
instr__ : & NATB \times RELB_{26} & \longrightarrow & REG & /*\ Jump,\ branches,\ 26\ bits\ imd\ */ \\
 & & & & \\
_ : & FLOAT & \longrightarrow & REG & /*\ FLOAT\ to\ REG\ conversion\ */ \\
_ : & RELB & \longrightarrow & REG & /*\ RELB\ to\ REG\ conversion\ */ \\
_ : & NATB & \longrightarrow & REG & /*\ NATB\ to\ REG\ conversion\ */
\end{array}
$$

$L = \{UNDEFINSTR,$ /* *unknown unspecified instructions* */
$TBSL\}$ /* *instructions specified later* */

The number of bits used to represent the instruction set allows to encode more than the DLX's 90 instructions. Consequently, some binary codes have no corresponding instruction; their label is "UNDEFINSTR".
"TBSL" (To Be Specified Later) is another label indicating that the instruction will be specified later (e.g. in another refinement step). Thus, the high level specification becomes more legible.

The set of general axioms is defined by:

GenAx:

Unused Code-op
$instr(0, succ^1(0), x, y, z) \in UNDEFINSTR$ /* 000001 *has no associated instruction* */
$instr(0, succ^5(0), x, y, z) \in UNDEFINSTR$ /* 000101 *has no associated instruction* */
$instr(0, succ^{8..15}(0), x, y, z) \in UNDEFINSTR$ /* 001000 *to* 001111 *have no associated instructions* */
...

/* *Instruction not yet specified* */
SLL /* *shift left logical* (000100) */
$instr(0, succ^4(0), x, y, z) \in TBSL$
$MULTF$ /* *floating − point multiplication* (000010) */
$instr(1, succ^2(0), x, y, z) \in TBSL$
...

The label "TBSL" prevents incomplete specifications. For example, let us take a "TBSL" floating-point unit; its label will only be removed if we finally want to specify a complex processor with floating-point and integer units. If we want to specify a simpler processor, the label will be kept. In both cases, specifications are complete.
The reason why the **SLL** (*shift logical left*) and **MULTF** instruction are labeled by "TBSL" is that it is easier and more legible to specify the shift operation using array of booleans rather than integers.

Remark: Let us notice that there is no specific DefAx part for the register signature. The reason why is that *Ok*-Axioms contained in registers specifications (i.e. natural numbers, integers and floating points) are sufficient.

5.1.3 The memorisation elements

There are two different memorisation elements in DLX:

- the external memory
- the register set

Within the high level specification, both behave in the same way as they are composed of a set of registers. So, we have only one global exception specification, called the "MEM" specification. The corresponding signature is defined by:

$S = \{MEM\}$ /* *Sort correponding to the extern memory and the register file* */			
$F =$			
$write___ :$	$REG \times MEM_{SIZE} \times NATB$	$\longrightarrow MEM_{SIZE}$	/* *write* REG *in* $MEM_{SIZE}[NATB]$ */
$read__ :$	$MEM_{SIZE} \times NATB$	$\longrightarrow REG$	/* *read the content of* $MEM_{SIZE}[NATB]$ */
$create :$	$NATB$	$\longrightarrow MEM_{NATB}$	/* *Creation of a memory of* $NATB$ *size* */
$L = \{OutOfMemory\}$			/* *memory access forbidden* */

The set of axioms is defined by:

GenAx:

$ind < 0 = true \Longrightarrow write(r, br, ind)\ \epsilon\ OutOfMemory$ / *write forbidden at negative index* */

$SIZE < ind = true \Longrightarrow write(r, br, ind)\ \epsilon\ OutOfMemory$ /* *the index exceed the memory size* */

$ind < 0 = true \Longrightarrow read(br, ind)\ \epsilon\ OutOfMemory$ /* *negative index* */

$SIZE < ind = true \Longrightarrow read(br, ind)\ \epsilon\ OutOfMemory$ /* *index too big* */

DefAx:

$write(r1, write(r2, br, ind), ind) = write(r1, br, ind)$ /* *Rewrite on a "REG" will overwrite value* */

$eq(ind1, ind2) = false$ /* *write order in two different cases is undiferrent* */

$\Longrightarrow write(r1, write(r2, br, ind2), ind1) = write(r2, write(r1, br, ind1), ind2)$

$read(write(r, br, ind), ind) = r$ /* *Read operation give the last written value* */

From now on, we can specify the execution of the instructions contained in the memory.

5.1.4 The instruction scheduling

The entity Execution Unit is quite special; it is not a real-life entity of the DLX processor. In our specifications, it represents a virtual execution unit, including different steps of the DLX's pipeline: *fetch*, *decode*, *execute*...

The instruction scheduling specification allows the introduction of dynamic aspects in the DLX specification. Let us give the specification of the Execution Unit:

$S = \{INSTR\}$	$/*$ *instruction scheduling sort* $*/$
$F =$	
$[_, _, _] : REG \times MEM \times MEM \longrightarrow INSTR$	$/*$ *static state* $(PC, GPR, Memory)$ $*/$
$exe_ : INSTR \longrightarrow INSTR$	$/*$ *go to the next state (next instruction)* $*/$
$L = \{ERRORINFETCH\}$	$/*$ *execution, instruction error* $*/$

The first operation represents a state of the DLX at a given step: this static state is composed of a memory, a register set and a counter program. The second one allows to go from one state to the next. In practice, the description consists in executing an instruction and fetching the next one. The label "ERRORINFETCH" is used to represent either a code operation which does not match any instruction or an error which occurs during the execution of an instruction.

GenAx:
$/*$ *propagation of errors occured in the register − register instruction format* $*/$
$instr(x, y, z, t, u) \,\epsilon\, TBSL$ $/*$ *Instruction specified latter* $*/$
$\Longrightarrow exe([pc, br, mem]) \,\epsilon\, ERRORINFETCH$
$instr(x, y, z, t, u) \,\epsilon\, UNDEFINSTR$ $/*$ *Undefined instruction co − op* $*/$
$\Longrightarrow exe([pc, br, mem]) \,\epsilon\, ERRORINFETCH$
... $/*$ *other instruction format* $*/$

The label "$ERRORINFETCH$" means that the execution of the instruction under consideration is either not yet implemented or an invalid instruction.
The exception raising "ERRORINFETCH" is caused by the propagation of exceptions occuring on previous terms of the sort "MEM".

The $DefAx$ part is defined by:

DefAx:

NOP $/*$ *no operation , just start the next instruction* $*/$
$read(mem, pc) = instr(0, 0, x, y, z)$ $/*\ 000000\ code-op\ */$
$\Longrightarrow exe([pc, br, mem]) = [pc + 1, br, mem]$ $/*$ *next instruction* $*/$

ADD $/*\ z\ =\ x +\ y\ */$
$\left.\begin{array}{l} read(mem, pc) = instr(0, succ^{32}(0), x, y, z) \\ (a = read(br, x)) \wedge (b = read(br, y)) \end{array}\right\}$ $/*\ code-op\ 100000\ */$ $/*$ *load x and y from the file register* $*/$
$\Longrightarrow exe([pc, br, mem]) = [pc + 1, write(a + b, br, z), mem]$ $/*$ *store the sum, next instrution* $*/$

LW $/*$ *load word from memory and store it in register* $*/$
$read(men, pc) = instr(succ^{35}(0), x, y, z))$ $/*\ code-op\ 100011\ */$
$\Longrightarrow exe([pc, br, mem]) = [pc + 1, write(read(mem, y + z), br, x), mem]$ $/*$ *read memory*$[y + z]$, *next instr* $*/$

SW $/*$ *load word from file register and store it in mem* $*/$
$read(men, pc) = instr(succ^{43}(0), x, y, z))$ $/*\ code-op\ 101011\ */$
$\Longrightarrow exe([pc, br, mem]) = [pc + 1, br, write(read(br, x), mem, y + z)]$ $/*$ *write in mem*$[y + z]$, *next inst* $*/$

$BNEZ$ $/*$ *conditional branch* $*/$
$\left.\begin{array}{l} read(men, pc) = instr(succ^{5}(0), x, y, z)) \\ eq(x, 0) = false \end{array}\right\}$ $/*\ code-op\ 000101\ */$ $/*\ case: value\ of\ register\ x\ != 0\ */$
$\Longrightarrow exe([pc, br, mem]) = [pc + z, br, mem]$ $/*$ *branch at* $pc + z$ *adress* $*/$

$\left.\begin{array}{l} read(men, pc) = instr(succ^{5}(0), x, y, z)) \\ eq(x, 0) = true \end{array}\right\}$ $/*\ code-op\ 000101\ */$ $/*\ case: value\ of\ register\ x\ == 0\ */$
$\Longrightarrow exe([pc, br, mem]) = [pc + 1, br, me]$ $/*$ *next instruction* $*/$

JR $/*$ *jump with target addressed by register* $*/$
$read(men, pc) = instr(succ^{2}(0), x))$ $/*\ code-op\ 000011\ */$
$\Longrightarrow exe([pc, br, mem]) = [read(br, x), br, mem]$ $/*$ *jump at address* x $*/$

$/*\ Where\ pc\ : NATB;\ a, b : RELB;\ */$

Let us notice that the pipeline is underlying:

- Fetch and decode of the instruction
  $read(men,pc) = instr(succ^{5}(0),x,y,z))$.
- Registers fetch
  $a = read(br,\ x)$
- Execution step and memory access. $exe[_,_,_] = [_,_,_]$

We do not consider it in this paper to simplify the DLX specification. Nevertheless, it is possible to describe such an architectural structure into our specification framework.

5.2 Low Level Specification of the DLX Processor

5.2.1 Abstract implementation

The aim of an abstract implementation is to specify a higher level abstract data type by another lower level data type and to give:

- an axiomatic specification describing *how* the lower level data types are used to implement (simulate) the higher level data type under consideration.
- semantics for this specification.
- means to prove its correctness.

The main advantage of such an approach is that the formalism used for all the specifications (the lower level data types, the higher level data type and the abstract implementation) is the same. Thus, they share the same proof techniques. Consequently, correctness proofs for abstract implementations are easier because they can be performed according to a homogeneous logic. We can prove that the implementation of the higher level specification by the lower level specification is correct. Consequently, the final specification behaves in the manner specified.

In this article, we use the [EKMP80]'s abstract implementation formalism extended to exception handling [Ber89]. In this framework, the *abstraction method* is used to algebraically specify the implementation. The aim of this method is to abstract tuples of lower level data types into higher level abstract data types. To reach this purpose, we use an abstraction operation, denoted by "$< \ldots >$".

From this approach, we have to consider two problems:

1. two tuples of lower level data may be abstracted into only one higher level abstract data (e.g. when we implement stacks by the couple $array \times nat$, all tuples $<t, 0>$ belonging to $array \times nat$ represent the same empty stack).
2. some tuples of lower level data implement nothing (e.g. from the same implementation than above, the tuple $<create, 4>$ belonging to $array \times nat$ where $create$ represents the uninitialized array, does not represent a possible stack because the four first places must be initialized).

As we will see below, in the [EKMP80]'s formalism, these difficulties are solved at semantic level.

5.2.2 Formalism presentation

The main idea of this formalism is to distinguish between the syntactical level and semantic level of implementations:

- At the syntactical level, first of all, we dispose of two exception specifications:

  1. the specification of the lower level data types, denoted by: $Spec_0 =< \Sigma_0, L_0, Gen_{Ax_0}, Def_{Ax_0} >$.
  2. the specification of the higher level data type that we want to implement, denoted by: $Spec_1 =< \Sigma_1, L_1, Gen_{Ax_1}, Def_{Ax_1} >$.

  Moreover, we can specify an intermediate hidden specification $Spec_H$.

  Thus, a syntactical implementation is defined by an tuple:

$$Impl =< F_{Abs}, Spec_H, Gen_{Ax_{op}}, Def_{Ax_{op}} >$$

  where:

  - F_{Abs} is the set of abstraction operations provided with a co-domain belonging to $S_1 \coprod S_H$.
  - $Spec_H =< \Sigma_H, L_H, GenAx_H, DefAx_H >$ is the specification describing the behavior of hidden operations of the implementation. These ones are only used to specify the implementation. They are not visible outside.
  - $GenAx_{op}$ (resp. $DefAx_{op}$) is the set of generalized axioms (resp. default axioms) describing the "constructive" implementation of higher level operations.

- The semantic level is composed by three successive applications: *synthesis*, *identification* and *restriction*[2].

  - the synthesis application turns each algebra satisfying the resident algebras (i.e. the specification $Spec_0$) into only one algebra satisfying the implementation (i.e. the specification *Impl*).
  - the identification application identifies all tuples which implement the same higher level data. Consequently, it allows to semantically solve the first difficulty mentioned above.
  - the restriction application prunes all tuples which implement nothing. So, it semantically solves the second difficulty mentioned above.

[2] As the class of algebras which satisfy a given specification, defines a *category* (for more detail, see [BW90]), the synthesis, identification and restriction applications are generally called *functors* (i.e. applications which preserve morphisms between algebras).

5.3 The DLX abstract implementation

What we intend to implement in DLX concerns data types described in the high level specification (natural numbers, integers and floating-point numbers). We use the array data type (see the ARRAY specification of Example 3.1 with the sort ELEM instantiated by BOOL) to implement bounded natural numbers in binary representation. Consequently, the only element of F_{Abs} is:

$$\boxed{<_>: Array \rightarrow NATB}$$

So, given an array t for which the size is n, the corresponding natural number val is obtained by the following formulas: $val = succ^n(0)$ where $n = \sum_{i=1}^{n} t[i] \times 2^i$.[3]

To implement bounded natural numbers from arrays, supplementary operations on the already implemented structures (i.e. *Array* and *Boolean*) are useful to specify the final implementation. The behavior of these operations is defined into the hidden specification of the implementation $Spec_H$. They are specified as follows:

$S = \{ARRAY, BOOL, NATB\}$ /* *Names of sorts* */			
$F =$			
$Wide:$		$\longrightarrow$	$NATB$
$unif___:$	$ARRAY \times BOOL \times NATB$	$\longrightarrow$	$BOOL$
$plus_____:$	$ARRAY \times ARRAY \times ARRAY \times BOOL \times NATB$	$\longrightarrow$	$ARRAY$

Intuitively, $unif(t, b, n)$ is true if all cells of t between 0 and n uniformly contain the value b, $Wide$ gives the size of arrays under consideration in the specification and $plus(t_1, t_2, t_s, ret, n)$ is the classical binary addition (using n full adders, ret is the carry) of the arrays t_1 and t_2 from the cell 0 to the cell n. The array t_s is the addition result.

For the sake clarity, we only specify the operation $plus$. So, we obtain the following axioms:

[3] Let us remark that the identification and restriction applications are only identity applications. The reason for it is that the abstraction operation $<_>$ is bijective (this property is easy to show). At each array we have only one natural number which is implemented by it. Conversely, at each natural number, it corresponds only one array which implemented it.

GenAx:

$/*$ *sum of numbers too big* $*/$

$$\left.\begin{array}{r}(t1[Wide-1]\ and\ t2[Wide-1])\\ or\ (carry\ and\ (t1[Wide-1]\ or\ t2[Wide-1]))\end{array}\right\} \Rightarrow plus(t1,t2,t3,carry,Wide)\ \epsilon\ Overflow$$

DefAx:

$/*$ *description of the sum with logical operators* $*/$

$plus(t1,t2,t3,carry,Wide) = store(t3,Wide,((t1[Wide]\ xor\ t2[Wide])\ xor\ carry))$

$plus(t1,t2,t3,carry,n) = plus(t1,t2,store(t3,n,((t1[n]\ xor\ t2[n])\ xor\ carry),$
$(t1[n]\ and\ t2[n])\ or\ (carry\ and\ (t1[n]\ xor\ t2[n]),succ(n))$

$/*$ *Where WIDE is the array size.* $*/$

Remark: The description into a hardware language such as **VHDL** is very close to the description above.

```
procedure plus  (t1, t2 : in bit_vector;
                 t3 : out bit_vector ) is
    variable result : bit_vector(t1'range);
    variable carry : bit := '0';
  begin
    for n in result'reverse_range loop
        result(n) := t1(n) xor t2(n) xor carry;
        carry := (t1(n) and t2(n)) or (carry and (t1(n) xor t2(n)));
    end loop;
    t3 := result;
  end plus;
```

Here, the abstract implementation of operations belonging to bounded natural numbers specification is defined by:

GenAx:

$unif(t,true,Wide) = true \Longrightarrow succ(<t>)\ \epsilon\ Overflow$ $\quad /*\ succ(0111...111)\ give\ an\ overflow\ */$

$succ(<t1>)\ \epsilon\ Overflow \Longrightarrow succ(<t1>) = <t1>$ $\quad /*\ succ\ do\ nothing\ on\ a\ too\ big\ number\ */$

$succ <t1> + <t2>\ \epsilon\ Overflow \Longrightarrow$ $\quad <t1> + succ <t2>\ \epsilon\ Overflow$

DefAx:

$unif(t,false,Wide) = true \Longrightarrow 0 = <t>$ $\quad /*\ element\ 0\ definition\ */$

$<t1> = 0 \Longrightarrow succ(<t> = <plus(t,store(t1,0,true),t2,false,0)>$

$succ(<t>) = <plus(t,store(<nul>,0,true),t2,false,0)>$ $\quad /*\ succ(n) = n + (000....001)\ */$

$<t1> + <t2> = <plus(t1,t2,t3,false,Wide)>$ $\quad /*\ integer\ sum\ conversion\ into\ bit\ array\ sum\ */$

$\cdots$ and so on for operations: $-, eq, <$

We assume that the maxint-value is represented by the array where each cell contains *true*. So, when we apply the operation *succ* on it, we obtain a higher value. Consequently, such a term has to be labeled by *Overflow*. The second axiom defines the exception handler. Lastly, as previously, the last two axioms describe general properties on every term.

The $DefAx$ part defines the constructive specification. It describes how the operations 0, *succ* and + of the higher level specification are implemented from resident specification operations, the hidden operation *plus* and abstraction operation $<_>$.

As previously, we do not consider integers and float implementation.

5.3.1 SLL instruction

As we mentioned before, the SLL instruction has not been specified in the high-level specification (this has been made possible by using the label "TBSL"). The reason for it is that it is easier to specify a shift with the ARRAY structure in the implementation than with RELB and NATB data types. Now, at this level of description, we get the following specification:

$F =$ /* *left logical shift* */

$sll\ ____ : \quad ARRAY \times ARRAY \times NATB \times NATB \longrightarrow ARRAY$

...

DefAx:

$0 < n - count = false$ /* *fill the "count" less significants bits with 0* */
$\Longrightarrow sll(< t1 >, < t2 >, count, n) = sll(< t1 >, < store(t2, n, 0) >, count, pred(n))$
$0 < n - count = true$ /* *shift the content of* $t[n - count]$ *to* $t[n]$ */
$\Longrightarrow sll(< t1 >< t2 >, count, n) = sll(< t1 >, < store(t2, n, t1[n - count]) >, count, pred(n))$
$sll(< t1 >, < t2 >, count, 0) =< t2 >$ /* *return t2 when all shifted* */

5.4 Implementation of the instruction set

This section is devoted to the implementation of the subset of the DLX instructions set, previously defined.
Let us assume that some constants like *nop, add, sll, j, null*[4] ... are specified in the hidden part of the implementation specification.

[4] *null* is the array where each cell is at $false$.

$DefAx:$

NOP
$mem[pc] = instr(nul,\ nop,\ <t1>,\ <t2>,\ <t3>) \Longrightarrow exe[<pc>,\
,\ <mem>]$
$\quad = [<pc> + <store(nul, 0, 1)>,\
,\ <mem>]$

ADD
$mem[pc] = instr(UE,\ add,\ <t1>,\ <t2>,\ <t3>) \Longrightarrow exe[<pc>,\
,\ <mem>]$
$\quad [<pc> + <store(nul, 0, 1)>,\ <\ store(br, <t3>, <t1> + <t2>)\ >,\ <mem>]$

SLL
$mem[pc] = instr(UE,\ sll,\ <t1>,\ <t2>,\ <t3>) \Longrightarrow exe[<pc>,\
,\ <mem>]$
$\quad = [<pc> + <store(nul, 0, 1)>,\ <store(br, <t3>, sll(<t1>,\ <t2>,\ <t3>,\ Wide)),\ <mem>]$

J
$mem[pc] = instr(j,\ <t>) \Longrightarrow exe[<pc>,\
,\ <mem>]$
$\quad = [<pc> + <t>,\
,\ <mem>]$

LW
$mem[pc] = instr(lw,\ <t1>,\ <t2>,\ <t3>) \Longrightarrow exe[<pc>,\
,\ <mem>]$
$\quad = [<pc> + <store(null, 0, 1)>,\ <store(br, <t3>, mem[<t1> + <t2>])>,\ <mem>]$

SW
$mem[pc] = instr(sw,\ <t1>,\ <t2>,\ <t3>) \Longrightarrow exe[<pc>,\
,\ <mem>]$
$\quad = [<pc> + <store(nul, 0, 1)>,\
,\ store(mem, <t2> + <t3>, <br[<t1>]>)]$

5.4.1 Implementation correctness

As mentioned before, the high-level, low-level and implementation specifications are written in the same formalism. Consequently, correctness proofs for abstract implementation are easier because they can be performed according to an homogeneous logic.

Intuitively, an implementation will be correct provided that the behavior of the implementation is indistinguishable from the behavior of the higher level specification under consideration.
Formally, [EKMP80] defines the correctness of an abstract implementation by the following two conditions:

1. The first condition, called *op-completude*, means that every ground term belonging to the higher level specification is (recursively) defined by the implementation. Consequently, such a condition means that every operation in higher level specification is completely specified. This condition is

easy to verify. It is often sufficient to inductively reason on operations belonging to the higher level specification. For example, our implementation of natural numbers by the arrays data type trivially verifies this condition. The reason is that the abstraction operation $<_>: Array \rightarrow Nat$ is surjective (in fact, as we have seen before, it is bijective).

2. The second one means that the implementation completely simulates the higher level specification. Such a condition means that we can prove every axioms belonging to the higher level specification from the axioms of the implementation.

   For example, let us prove the first axiom of the $DefAx$ part in the natural numbers specification:

   $$x \ + \ 0 = x$$

   (i.e. 0 is neutral for the sum operation).

   As already seen, the abstract operation $<_>: ARRAY \rightarrow NATB$ is bijective. Consequently, there exists an unique operation $\rho : NATB \rightarrow ARRAY$ (we have no overflow problems because $x \ + \ 0 = x$ is a $DefAx$-axiom). So, it is sufficient to prove the following formula:

   $$\rho(x) + 0 = \rho(x)$$

   By the first axiom of the natural numbers implementation, we can write:

   $$\rho(x) \ + \ \rho(0) = <plus(\rho(x), \rho(0), t, false, 0)>$$

   Consequently, it is sufficient to prove the following property:

   $$\forall i \in [1, Wide], \ plus(\rho(x), \rho(0), t, false, 0)[i] = \rho(x)[i]$$

   To reach this result, inductively show on the following lemma:

   Lemma:

   $$\forall n \leq Wide, \ \forall i \in [1, n], \ plus(\rho(x), \rho(0), t, false, 0)[i] = \rho(x)[i] \quad \text{and } carry = false \text{ (at the step } n)$$

   Proof:

   Basic case: At the first step of calculation, we have to consider two cases:

- if $Wide = 0$ then by the first $DefAx$-axiom of the hidden specification, we can write:

$$plus(\rho(x), \rho(0), t, false, 0) = store(t, 0, \rho(x)[0])$$

So, by the second axiom of the array specification, we can write:

$$store(t, 0, \rho(x)[0])[0] = \rho(x)[0]$$

Consequently, we directly conclude:

$$plus(\rho(x), \rho(0), t, false, 0)[0] = \rho(x)[0]$$

- if $Wide \neq 0$ then by the second axiom of the hidden specification, we can write:

$$plus(\rho(x), \rho(0), t, false, 0) = plus(\rho(x), \rho(0), store(t, 0, \rho(x)[0]), false, succ(0))$$

Let us remark that from the hidden specification, it is easy to prove:

$$\forall i \in [1, n-1],\ plus(t1, t2, t3, carry, n)[i] = t3[i]$$

Consequently, by the second axiom of the array specification, we can directly write:

$$plus(\rho(x), \rho(0), t, false, 0)[0] = \rho(x)[0]$$

In the same way, it is obvious that the resulting carry is $false$.

General case: Let us assume that t be the array obtained to the nth step of calculation. According to the induction hypothesis, we have:

$$\forall i \in [1, n],\ t[i] = \rho(x)[i] \text{ and } carry = false$$

As the basic case, concerning the $n+1^{th}$ step, we have to consider two cases:

- if $Wide = n+1$ then by the first $DefAx$-axiom of the hidden specification, we can write:

$$plus(\rho(x), \rho(0), t, false, n+1) = store(t, n+1, \rho(x)[n+1])$$

Moreover, by the second axiom of the array specification, we also have:

$$store(t, n+1, \rho(x)[n+1])[n+1] = \rho(x)[n+1]$$

As we have seen before, let us remember that:

$$\forall i \in [1, n-1],\ plus(t1, t2, t3, carry, n)[i] = t3[i]$$

Consequently, we directly conclude:

$$\forall i \in [1, n+1],\ plus(\rho(x), \rho(0), t, false, 0)[i] = \rho(x)[i]$$

- if $n+1 < Wide$ then by the second axiom of the hidden specification, we can write:

  $$plus(\rho(x), \rho(0), t, false, n+1) = plus(\rho(x), \rho(0), store(t, n, \rho(x)[n]), false, succ(n+1))$$

  Consequently, we can directly write:

  $$\forall i \in [1, n+1],\ plus(\rho(x), \rho(0), t, false, 0)[i] = \rho(x)[i]$$

  In the same way, it is obvious that the resulting carry is $false$.

6 Conclusion

The work partially reported in this paper shows that property oriented specifications are also well suited to specify Hardware. Similar results prepare the way to a new approach for Hardware/Software co-design, using formal methods and axiomatic specifications. In particular, we have shown that proving techniques of algebraic specifications and stepwise refinements can be achieve in the same manner as in Software.

This article reports a case study of Hardware specification using algebraic specification with exception handling. We have shown on this example how axiomatic specifications fulfill the fundamental constraints of co-design:

- to be in position to specify without hypotheses about future implementations (Hardware / Software choices).
- to be in position to specify the Hardware and Software parts without a priori management of the system complexity (i.e. flexible design of Hardware and Software parts). This offers at the designer the possibility to make correctness proofs of his or her system as soon as possible in the design process. Consequently, the mistakes of specifications fraught with consequences are avoided.

Let us note that this properties directly follow from using of axiomatic specifications.

This case study convinced ourselves of the importance of exception handling and modularity. These two concepts are crucial to reach a legible and terse specification style. Modularity also provides reusability of specification modules. It would be of first interest to enrich classical software specifications libraries with specific hardware modules.

For general purpose system specifications, it seems to be necessary to complete a modular approach with concurrent and dynamic aspects between modules. With this respect, the potential benefits of Object Oriented concepts have been illustrated in [AJKW94], but at a non formal level, using the C++ language. Algebraic specifications with Object Oriented aspects belong to the latest issues of the European ESPRIT working group IS-CORE [AB95]. The goal of our future works is to extend these developments to Hardware/Software co-specification. Indeed, such a formalism offers new perspectives for the co-design as complex interactions between elements of a system and implicit dynamic aspects (states, temporality notion, ...). The main advantage of such a formalism is that resulting specifications are more abstract , clearer and terser. Thus, this allows to delay implementation steps.

Acknowledgements

We would like to thank Judith Benzakki for careful proof readings of this paper and accurate corrections.

References

[Abr94] J-R. Abrial, : "*Assigning meaning to programs*", to be published, Cambridge University Press, 800 pages, 1994.

[AB95] M. Aiguier, G. Bernot, : "*Algebraic semantics of object type specifications*" to be published in proceedings of IS-CORE'94, World Scientific Publishing, 1995.

[ABBI94] M. Aiguier, J. Benzakki, G. Bernot, M. Israël, : "*Ecos: From Formal Specification to Hardware / Software Partitionning*", VHDL Forum'94, Grenoble, France, 1994.

[AJKW94] H.A. Aylor, B.W. Jonhson, S. Kumar, W.A. Wulf : "*Object-Oriented Techniques in Hardware Design*", Computer Science, pp. 64-70, June 1994.

[Ber89] G. Bernot, :"*Correctness proofs for abstract implementations" Information and Computation (formerly Information and Control)*", Vol.80, No.2, pp.121-151, February 1989.

[BGA94] G. Bernot, P. Le Gall, and M. Aiguier, : "*Label algebras and exception handling*" *Science of Computer Programming (23)*, North-Holland Pub., pp. 227-286, 1994.

[BGM89] M. Bidoit, M-C. Gaudel and A. Mauboussin, : "*How to make algebraic specifications more understanble? an experiment with the PLUSS specification language*", Science of Computer Programming, 12(1), 1989.

[BH85] B. Biebow, and J. Hagelstein : "*Algebraic specification of synchronization and errors: A telephonic example*", In Proc. of the 1st International Joint Conference on Theory and Practice of Software Development (TAPSOFT), pp. 294-308, Springer-Verlag, L.N.C.S. 186, 1985.

[BW82] M. Broy and M. Wirsing, : "*Partial abstract data types*", Acta Informatica, 18(1), November 1982.

[BW90] M. Barr, and C. Wells, : "*Category Theory for Computing Science*", Prentice-Hall, 1990.

[Cap87] F. Capy, : "*ASSPEGIQUE : un environnement d'exceptions... Une sémantique opérationnelle des E,R-algèbres, formalisme prenant en compte les exceptions. Un environnement intégré de spécification algébrique: ASSPEGIQUE*", PhD thesis, Université de Paris-Sud, décembre 1987.

[DG94] P. Dauchy, and M.C. Gaudel, : "*Algebraic Specifications with implicit state*", LRI, Université de Paris-Sud, Tech. report n.887,1994.

[DS83] J. Despeyroux-Savonitto, : "*An algebraic specification of a Pascal compiler*", SIGPLAN notices, 18(2), 1983.

[EKMP80] H. Ehrig, H. Kreowski, B. Mahr, P. Padawitz, : "*Algebraic implementation of abstract data types*", Theoritical Computer Science, October, 1980.

[EM85] H. Ehrig and B. Mahr, : "*Fundamentals of algebraic specification 1. equations and initial semantics*", EATCS Monograph on Theoteritical Computer Science, 6, 1985.

[GTW78] J.A. Goguen, J.W. Thatcher, and E.G. Wagner, : "*An initial algebra approach to the specification, correctness, and implementation of abstract data types*" In Current Trends in Programming Methodology, R.T. Yeh Prentice-Hall éditeur, volume IV, pp.80-149, 1978. Also IBM Report RC 6487, October 1976.

[Gog78] J.A. Goguen, : "*Order sorted algebras: exceptions and error sorts, coercion and overloading operators*" Tech. report n.14, University of California Los Angeles, Semantics Theory of Computation, December 1978.

[HP90] J.L. Hennessy, and D.A. Patterson, : "*Computer Architecture: A Quantitative Approach*" Morgan Kaufmann Publishers, Inc, 1990.

[HP94] J.L. Hennessy, and D.A. Patterson, : "*Computer Organization and Design. The Hardwarw / Software Interface*" Morgan Kaufmann Publishers, Inc, 1990.

[Jon90] C.B. Jones, : "*Systematic Software Development using VDM*", Prentice Hall International, 2nd edition, 1990.

[KB70] D-E. Knuth and P-B. Bendix, : "*Simple word problems in universal algebras*", In J. Leech, ed., Computational Problems in Abstract Algebras, pp. 263-297, Pergamon Press, Oxford, U.K., 1970. (reprinted [1983] in Automation of reasonning 2, Springer, Berlin, pp. 342-376.)

[Mos89] P. Mosses, : "*Unified algebras and action semantics*", In Proc. STACS'89, 6th Symp. on Theoritical Aspects of Computer Sciences, February 1989.

[MSS89] V. Manca, A. Salibra and G. Scollo, : "*Equational type logic*", In Conference on Algebraic Methodology and Software Technology, pp. 131-159, May 1989. Iowa City, Iowa (TCS 77).

[Spi92] J.M. Spivey, : "*The Z notation: a reference manual*", Prentice Hall International, 2nd edition, 1992.