On Teaching the Concept of Refinement with B

Joanna Tomasik, Guy Vidal-Naquet

On Teaching the Concept of Refinement with B

Joanna Tomasik Guy Vidal-Naquet

Computer Science Department
SUPELEC
Gif-sur-Yvette, France

Abstract

The concept of refinement is central to the development of software. It appears in various forms in the different methodologies taught to students. A key point in the B method is the validation of the refinement step. The B methodology exhibits mathematical properties of correct refinements, and also automatically checkable conditions that ensure those properties. Some of the main pedagogical difficulties that the present authors found in teaching B centered around the notions linked to refinement, at the conceptual level, and at the tool level. Many papers have been published on the general benefits of the B method. This paper will focus on the specific concepts linked to refinements, and on the ones which need special care. We argue that, although B presents a complete mathematical analysis, it is beneficial to put the concept of refinement in perspective with other theories that come from formal methods, namely, in this paper, coalgebra and bisimulation.

Keywords: Coalgebra, Gluing invariant, Morphism, Refinement.

1 Introduction: pedagogical aspects linked to positioning B in a software engineering cursus

The necessity of a methodology for producing quality software is not questioned nowadays. Such was not the case a few years ago, as shown by a study of the Software Engineering Institute (SEI) [12]. To our knowledge, this is the last available diagram that the SEI published which presents the evolution per year. It shows that during the years 1987–1991 more than eighty percent of software companies were at the initial or ”chaotic” level of maturity. As one can see on the diagram in Fig. 1, in 2005, a few companies were still at this level. Nevertheless this diagram shows that the maturity level of firms has increased. One of the acknowledged reasons is the fact that engineers who have followed a cursus in software engineering and its different methods, including formal methods, are working now in the software development industry.

[1] Email: Joanna.Tomasik@supelec.fr
[2] Email: Guy.Vidal-Naquet@supelec.fr

We think that the B method is an adequate answer to the old (but still mentioned) joke:

Formal methods have been, are, and will always be, the future of software development.

and that it will help increase the figures in the rightmost sections of this diagram.

One of the first questions that arises with teaching B is whether it should be integrated into a general course on software engineering, or be taught in a separate course. In view of the conceptual richness of the method, and of the need for having practical development by students in order to show the usefulness of this methodology, we definitely opted for the second solution.

UML occupies a central position in the teaching of software engineering. Some semantic meanings can be associated with the graphical syntax of UML. Rational Rose is a tool that is often associated with RUP (Rational Unified Process) and UML. A recurrent problem encountered when teaching UML with Rose is that of coherence between the different phases of processes of development (for instance, with Rose, between Case View and Logical View). Students ask questions about possible contradictions which may occur in a developed system seen on different process stages without being detected. They are truly disappointed when learning that there is not any formal method for detecting inconsistencies. They are also perturbed by the lack of a rigorous methodology for the construction of UML diagrams.

In addition, the use of mathematically founded methods, such as Hoare logic, and the corresponding system of proof seems to students to be distant from real applications. Students are supposed to make Hoare proofs manually. The conclusion they draw from this activity, however, is that these proofs cannot be performed automatically and are unapplicable in real cases.

For these reasons, software engineering teachers are confronted with a quandary:

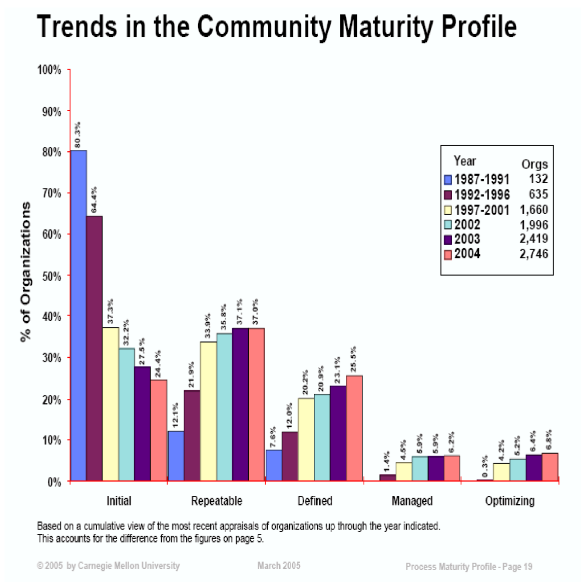

Fig. 1. Evolution of the maturity level

the need to use formal methods, and a lack of pedagogically effective ones. One important aspect of teaching B is that this apparent paradox disappears because the B methodology presents many concepts which have direct applications for the practical development of software.

In order to obtain a coherent curriculum, the teaching of B has to be related to the other courses on software engineering and formal methods. The common and specific aspects of B must be addressed not only in the course on B, but also in other courses. This means that in other courses mention of the way that B deals with specific questions, i.e. encapsulation, should be brought out, and in the specific course on B, the relationship with other methodologies should be examined.

Teaching the B method illustrates how the utilisation of formal methods contributes to software development's being an industrial activity. While important concepts, such as preconditions and offensive programming, are easily understood by students, we found that the real difficulties center around the notion of refinement and the meaning of the obligations of proof that are generated. In this paper, we will concentrate on the questions raised by refinement - not the concepts themselves, but on those aspects students find difficult and/or useful and on how to address these difficulties.

In our teaching, we use the AtelierB tool. We chose this tool, as opposed to B4free — available on the Internet without any licence — for several reasons. We want students to become familiar with the software that is used in the industry, but the most important reason for our choice is the possibility of generating, compiling and executing code in a programming language. We teach in an engineering school and our students are conscious of their future professional career development. They want to "see the things working".

We observed that students work seriously during classes concerning abstract machine definitions, refinement and project structure. The class in which students obtain their first executable programs leads to real enthusiasm. The difference in the attitude of the students before and after the obtention of executable code is striking. They recognize that all their efforts to reach an appropriate expression of logical properties lead to an executable program. They see this as a concrete outcome of the B development process. This observation makes them strongly motivated to progress in the B method.

Our students work on their individual projects, which have to be seen in an overall context. Students are asked to write a specification of a proposed system and follow the B development process to the end, i.e. an executable program, while paying attention to reducing the number of proof obligations that are not automatically proven. Due to the number of hours devoted to B, we cannot present the interactive prover in detail, but we insist that the students comment on the origin of the proof obligations generated by AtelierB. All of our students show a deep interest in having the best possible project and they invest a lot of time and energy in order to succeed. We are therefore convinced that the goal of obtaining the best code for an executable program stimulates them.

One of the aspects that students appreciate in the B methodology is the continuity in the development process, and, as mentioned before, the integration of formal methods. Predicate logic and the theory of proof system blend nicely with their concrete development activities. Some residual questioning about the interest of the method and negative feedback are linked to the refinement step.

We found that beneficial effects result from spending more time on explaining the relationship between the states of a machine and the states of a refinement machine, as well as from formalizing the concept expressed informally by "a machine behaves like another". In order to do that in depth, we propose to make the links and differences between refinement and coalgebra theory explicit in the teaching cursus. We firmly believe that *THE* method does not exist, and that it is the duty of a teacher to show the relationship of what she/he teaches with the other aspects of a field.

Much of what we say is a rewriting in coalgebraic terminology of definitions and results present in the B methodology. This fact strengthen the ideas of the existence of links between the two approaches, and we believe that these links should be made explicit to the students.

The paper is organized as follows after this introduction. In Section 2, we present the fundamental concepts of coalgebas and morphisms between coalgebras. We do not enter into mathematical details, and we hope that this section will show the interest of this emerging field. In Section 3, we present the links between B machines and coalgebra. In Section 4, we study the link between refinement, gluing invariants and morphisms of coalgebras. Finally, in Section 5, we discuss methods for teaching coalgebra theory in relation with the B methodology.

2 Basic concepts on coalgebras: observation versus construction

Recently, a considerable amount of work on coalgebras was done, in order to provide a mathematical basis for the concept of observation in state based systems. The study of coalgebra was spurred by works on bisimulation for parallel systems. See [10] for a seminal paper on the use of coalgebras and their relationship with objects. The papers [7] and [8] give good, pedagogical introductions to the concepts and main results on coalgebras, with a short survey on relevant definitions of the theory of categories. In [3] coalgebras are used for defining the semantic of the Rosetta language. We found the concepts on coalgebras very well suited for teaching the different notions of refinement in a unified way.

Algebraic definitions will tell how a system is *built*, while coalgebraic definitions will tell how a system is *observed*. Quite often these two aspects are mixed. For example consider the specifications for lists. There will be

- a constuctive part: how to build a list from the empty list, adding an element,
- an observation part: i.e. test for emptiness, length, last element.

A basic concept for a system that is used (but has not necessarily been developed by the user) is that one does not know how the possible states are obtained, but from a given state one can get some information. It is the set of states and the correspondence between a state and the information on this state that defines a given coalgebra. This is also the approach followed for object-oriented software engineering.

For example, let us consider a traffic light system. The internal mechanism can consist only of a timer, so the light changes at regular time interval, or, in addition, it can use a car detection system. The user of the system ignores its internal mechanisms and can just observe the color of the light. In this example, we associate to a set of internal states S (which is not specified) a simple structure consisting of the set $L = \{\text{green}, \text{yellow}, \text{red}\}$ and we associate to each element of S an element of L.

More generally, we can associate to any set X of states of a system — *the carrier set* — a more complex structure $F(X)$ that possibly involves X. The structure $F(X)$ indicates what we can know about a state in X. The operation F that associates to any carrier set the structure of what can be observed is called a *signature*. Mathematically F is a functor, but this can be ignored for our purpose. A coalgebra $\langle S, c\rangle$ is a realisation of a signature F. S is a set which is a concrete "instance" of X and c is the observation function from S to $F(S)$.

Definition 2.1 Given a signature F, a coalgebra for F is a couple $\langle S, c\rangle$ where S is a carrier set of states and c is an observation function from S into $F(S)$.

Example 2.2 Consider the following signatures

(i) $F_1(X) = L$, with $L = \{\text{green}, \text{yellow}, \text{red}\}$.
A coalgebra $\langle S, c\rangle$ for F_1 is defined by $S = \mathbb{N}$ and c in defined as follows:
- $\forall n, 0 \leq n \bmod 30 \leq 15$, $c(n) = \text{green}$
- $\forall n, 16 \leq n \bmod 30 \leq 18$, $c(n) = \text{yellow}$
- $\forall n, 19 \leq n \bmod 30 \leq 29$, $c(n) = \text{red}$

How the state is determined is irrelevant here. It could be done with the help of an egg timer, an atomic watch or a deck of cards.

(ii) $F_2(X) = X$.
For a coalgebra $\langle S, c\rangle$, the observation of a state s is a state $c(s)$.
Generally, the interpretation of c is the next state, but it could be the preceding state, or a state that we want to avoid, or a state that we want to reach.
For example, $S = \{u, v, t, w\}$ the observation function c is given by $c(u) = v$, $c(v) = w$, $c(t) = t$, $c(w) = v$.

(iii) $F_3(X) = \{\perp\} \cup X$.
For a given coalgebra $\langle S, c\rangle$, the observation of a state s is a state $c(s)$, or the element $\perp$. Generally, the interpretation of $c(s)$ is the next state, if $c(s) \in S$. When s is a state that cannot evolve (a final state), $c(s) =\perp$.

(iv) $F_4(X) = \{\perp\} \cup (X \times A)$.
For a coalgebra $\langle S, c\rangle$, the observation of a state s is a set of pairs composed of a state $c(s)$ and a label in A, or the terminal element $\perp$. The usual interpretation is that if the system can evolve, then it is possible to determine in which state it will be and also the label attached to the transition from one state to another. This coalgebra describes a deterministic labelled transition system.

(v) $F_5(X) = \{\perp\} \cup P(X \times A)$,
where $P(X)$ is the set of all subsets of X. For a coalgebra $\langle S, c\rangle$, the observation of a state s is a set of pairs composed of a state $c(s)$ and a label in A, or

the terminal element $\perp$. This coalgebra describes a non-deterministic labelled transition system.

(vi) $F_6(X) = \{0,1\} \times X^A$.
It is the signature associated with automata: given a state, the function of A into S is given, and one can also know if the state is final or not.

The notion of morphism is central in mathematics. It captures the intuitive ideas of an operation that preserves structures. For coalgebras, we have the following definitions:

Definition 2.3 Let $\langle S, c\rangle$ and $\langle T, d\rangle$ be two coalgebras for the same signature F.

If m is a function from S into T, $F(m)$ is the function that extends m to an function from $F(S)$ into $F(T)$. Informally $F(m)$ is obtained by replacing elements of S by their image in the expression defining an element of $F(S)$.

For example, if — for two carrier sets $S = \{s, s'\}$, $T = \{t, t'\}$ and two label sets A, B such that $a \in A$, $b \in B$ — F is the signature defined by $F(S) = (S \times A) \times (S \times B)$, and $m(s) = t$, $m(s') = t'$, then $F(m)(((s,a),(s',b))) = ((t,a),(t',b))$.

A function m is a morphism from S into T, when it respects the observation functions: $d \circ m = F(m) \circ c$.

In other words, the function m is a morphism when the following diagram commutes:

$$\begin{tikzcd}
F(S) \arrow[r, "F(m)"] & F(T) \\
S \arrow[u, "c"] \arrow[r, "m"'] & T \arrow[u, "d"']
\end{tikzcd}$$

Example 2.4 We follow the usual mathematical notation, $P(X)$ is the set of all subsets of X.
$F(X) = P(X \times \{a, b\})$
Let $S = \{s_1, s_2, s_3\}$, $c(s_1) = \{(s_2, a), (s_3, a)\}$, $c(s_2) = \{(s_3, b)\}$, $c(s_3) = \{(s_3, b)\}$,
$T = \{t_1, t_2\}$, $d(t_1) = \{(t_2, a)\}$, $d(t_2) = \{(t_2, b)\}$.

$\langle S, c\rangle$, and $\langle T, d\rangle$, can be graphically represented by the labelled transition systems, with m represented by the dotted line arcs (Figure 2). It is easy to check that the function $m(s_1) = t_1$, $m(s_2) = m(s_3) = t_2$ is a morphism. For example, starting from s_1 we have: $m(s_1) = t_1$, $d(m(s_1)) = \{(t_2, a)\}$. On the other hand: $c(s_1) = \{(s_2, a), (s_3, a)\}$, $F(m)(c(s_1)) = \{(t_2, a)\}$.

A main result which we will not develop here, but which is central to the coalgebra theory, is that if F is a "reasonable" signature then there exists a *final coalgebra* $\langle T, b\rangle$ for F, meaning that for any coalgebra $\langle S, c\rangle$ for F, there is a unique morphism from S into T. Intuitively, $\langle T, b\rangle$ represents all observations that can be made with the signature F

We refer to [8] or [7] for more details.

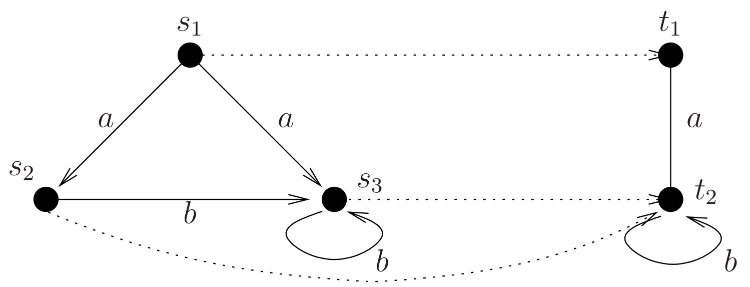

Fig. 2. Example of morphism between two coalgebras

A morphism from S to T corresponds to a deterministic way to establish a correspondence from S to T: given a state in S, one state in T is obtained. In general such a correspondence is non deterministic, i.e. to one object in S there correspond many objects in T. For example consider a LIFO queue implemented with an array and a number (the top of the stack). Intuitively a relation R between two sets S and T is a bisimulation for the signature functor F, when R "is carried through" by F. The formal definition is:

Definition 2.5 Let F be a signature, $\langle S, c\rangle$, $\langle T, d\rangle$ two coalgebreas for F, and $c : S \longrightarrow F(S)$, $d : T \longrightarrow F(T)$. A relation $R \subseteq S \times T$ is a *bisimulation* when there exists a coalgebra $\langle R, g\rangle$ such that the two projections $\pi_1 : R \longrightarrow S$ and $\pi_2 : R \longrightarrow T$ are coalgebra morphisms i.e. the following diagram commutes (Figure 3).

$$\begin{tikzcd}
F(S) & F(R) \arrow[l, "F(\pi_1)"'] \arrow[r, "F(\pi_2)"] & F(T) \\
S \arrow[u, "c"] & R \arrow[l, "\pi_1"] \arrow[r, "\pi_2"'] \arrow[u, "g"'] & T \arrow[u, "d"']
\end{tikzcd}$$

Fig. 3. Bisimulation of coalgebras

Example 2.6 Let $S = \{s_1, s_2, s_3\}$, $c(s_1) = \{(s_3, a)\}$, $c(s_2) = \{(s_3, a)\}$, $c(s_3) = \{(s_3, b)\}$, and $T = \{t_1, t_2, t_3\}$, $d(t_1) = \{(t_2, a), (t_3, a)\}$, $d(t_2) = \{(t_3, b)\}$, $d(t_3) = \{(t_3, b)\}$ (Figure 4).

The relation defined by $R = \{(s_1, t_1), (s_2, t_1), (s_3, t_2), (s_3, t_3)\}$ is a bisimulation.

Consider the function g from R into $F(R)$ defined by

$$g((s_1, t_1)) = \{((s_3, t_2), a), ((s_3, t_3), a)\},\ g((s_2, t_1)) = \{((s_3, t_2), a), ((s_3, t_3), a)\},$$
$$g((s_3, t_2)) = \{((s_3, t_3), b)\}, \qquad g((s_3, t_3)) = \{((s_3, t_3), b)\}.$$

It is easy but tedious to check that π_1 is a morphism from $\langle R, g\rangle$ into $\langle S, c\rangle$ and π_2 is a morphism from $\langle R, g\rangle$ into $\langle T, d\rangle$. In Figure 4, $\langle S, c\rangle$ and $\langle T, d\rangle$ can be viewed as two labelled transition systems, with the relation R shown by the dotted line. The notion of bisimulation in coalgebras is a generalization of the equivalent notions for CCS [9] and labelled transition systems [2]. The notion of simulation will play a central role in the way we present refinements.

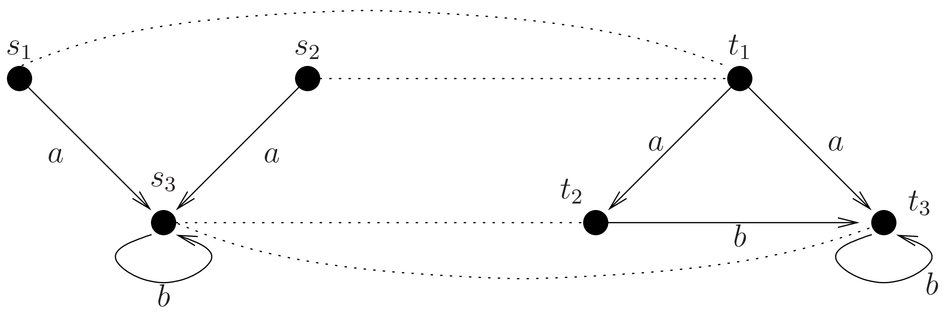

Fig. 4. Example of bisimulation between two coalgebras

When coalgebras correspond to non-deterministic labelled transitions systems (cf [2]), we have the notation $s \xrightarrow{\alpha} s'$ when $(\alpha, s') \in c(s)$ ($(\alpha, s') = c(s)$ in a deterministic case). For labelled transition systems, bisimulation corresponds to the fact that: for two labelled transition systems LS and LT with carrier states S and T, a relation R between S an T is a bisimulation when

If sRt then

(i) for all s' such that $s \xrightarrow{\alpha} s'$, there exists t', such that $t \xrightarrow{\alpha} t'$ and $s'Rt'$,

(ii) for all t' such that $t \xrightarrow{\alpha} t'$, there exists s', such that $s \xrightarrow{\alpha} s'$ and $s'Rt'$.

When only condition (i) holds, we say that R is a simulation of LS by LT. Thus we can see a simulation as "half of a bisimulation". This notion of simulation versus bisimulation can be extended to coalgebras. We will not go into the mathematical details about this and we refer to [11].

3 Links between B machines and coalgebras

The correspondence between B machines and signatures of coalgebras is quite natural. The notion of signature exists in B. We find it worthwhile to mention to the students the links between the B methodology and coalgebras. It helps convincing them that formal methods adress real problems and that there is not a unique way for decribing a method in mathematical terms.

We give an example with a machine with two operations `op1` and `op2`, where `op1` has a parameter and produces a result, while `op2` just changes the internal states which can easily be generalized. Consider the machine:

```
MACHINE MA
VARIABLES XX
INVARIANT IXX
INTIALISATION Z OPERATIONS
res <-- op1(NN) = PRE INN & C1 THEN S1 END;
op2 = PRE C2 THEN S2 END
END
```

The expression *IXX* is the invariant that defines the type `TXX` of the variable `XX`, *INN* is the expression that defines the type `TNN` of the parameter `NN` and `TRes` is the type of the result `res` of the operation. The signature of the coalgebra asssociated with the machine `MA` will be $F(X) = (\{\texttt{op1}\} \times (\{\perp\} \cup P(\texttt{TRes} \times X)^{\texttt{Tpar}})) \cup (\{\texttt{op2}\} \times$

$(\{\perp\} \cup P(X)))$. This signature allows us to observe sequences of operation executions and its final colgebra is a string. We consider $P(\mathtt{TRes} \times X)^{\mathtt{Tpar}}$ and $P(X)$ instead of $(\mathtt{TRes} \times X)^{\mathtt{Tpar}}$ and X, respectively, since *S1* and *S2* can be non deterministic.

Another possible signature could be $F(X) = (\{\mathtt{op1}\} \times (\{\perp\} \cup P(\mathtt{TRes} \times X)^{\mathtt{Tpar}})) \times (\{\mathtt{op2}\} \times (\{\perp\} \cup P(X)))$. This signature allows us to observe results of every operation for a given state and its final coalgebra is a tree.

A coalgebra for F is obtained when S corresponding to a set of the states defined by the invariant *IXX* and the observations is defined by the machine.

If a given state `XX` of the machine does not satisfy the precondition *C1*, one can observe that `op1` cannot be called and $c(\mathtt{XX}) = (\mathtt{op1}, \perp) \times \cdots$. If `op1`, whose parameter is in `Tpar`, is executed for a state `XX` then the machine passes to a new state i.e. the execution of *S1* attributes a new value to `XX` (which can be the same as the previous one) and the result is returned.

Let note that a signature does not give complete information on what the machine does, just the structure of what is produced. It depends on the observation one wants and the degree of knowledge on the process that one wants to observe. For example, if we are not interested in which operation is called, then for a machine having the two operations

```
res1 <-- op1(NN) = ...
res2 <-- op2(MM) = ...
```

If these two operation are deterministic, return results of the same type `TRes`, and their arguments have the same type `Tpar` then instead of the signature $F(X) = (\{\mathtt{op1}\} \times (\{\perp\} \cup (\mathtt{TRes} \times X)^{\mathtt{Tpar}})) \cup (\{\mathtt{op2}\} \times (\{\perp\} \cup (\mathtt{TRes} \times X)^{\mathtt{Tpar}}))$, it is possible to consider the signature $F(X) = \{\perp\} \cup (\mathtt{TRes} \times X)^{\mathtt{Tpar}}$. If an operation has no precondition (or a precondition equivalent to `TRUE`), then it is possible to have a signature of the form $F(X) = (\mathtt{TRes} \times X)^{\mathtt{Tpar}}$.

In the case of non-determinism of these opertations we replace $(\mathtt{TRes} \times X)^{\mathtt{Tpar}}$ by $P(\mathtt{TRes} \times X)^{\mathtt{Tpar}}$.

These exemples show that signature of coalgebras expresses what one **chooses** to observe.

Similarly to choosing the coalgebra associated with the machine, one can choose the set defined by the typing invariant and/or the set of reachable states induced by the initialisation and the operations.

4 Refinement, morphisms, and bisimulations

Consider the machine taken from [1]:

```
MACHINE M1
VARIABLES yy
INVARIANT yy:FIN(NAT1)
INITIALISATION yy:={}
OPERATIONS
enter(nn) = PRE nn:NAT1 THEN yy:=yy\/nn END;
mm <-- maximum = PRE yy /= {} THEN mm:=max(yy) END
END
```

The carrier set, defined by the invariant, is equal to $S = \texttt{FIN(NAT1)}$. The signature that is naturally associated with this machine is $F(X) = (\{\texttt{enter}\} \times X^{\texttt{NAT1}}) \times (\{\texttt{maximum}\} \times \texttt{NAT1})$. The coalgebra associated is $\langle S, c\rangle$ with $S = \texttt{FIN(NAT1)}$ and $c(s) = \{\texttt{enter}\} \times f_s \times \{\texttt{maximum}\} \times \texttt{max}(s)$, if $s \neq \emptyset$), where $f_s : \text{NAT1} \longrightarrow S$ defined by $f_s(n) = s \cup \{n\}$.

Now, consider this second machine also taken from [1]:

```
REFINEMENT R2
REFINES M1 VARIABLES zz
INVARIANT zz:NAT
INITIALISATION zz:=0

OPERATIONS
enter(nn) = PRE nn:NAT1 THEN zz:=max({nn,zz}) END;
mm <-- maximum = PRE zz /= 0 THEN mm:=zz END
END
```

We can associate the same signature F to this machine and the coalgebra $\langle T, d\rangle$ with $T = \texttt{NAT}$ and $d(z) = (\{\texttt{enter}\} \times g_z) \times (\{\texttt{maximum}\} \times (\perp \text{ if } z = 0,\ z \text{ if } z \neq 0)$, where $g_z : \texttt{NAT1} \longrightarrow T$ is defined by $g_z(n) = \max(n, z)$.

Consider the function m from the states S of M1 into the states T of R2 defined by $m(s) = 0$ if $s = \emptyset$ and $m(s) = \max(s)$ if $s \neq \emptyset$.

The initialisation of M1 produces the state where $\texttt{yy} = \emptyset$ and initialization of R2 produces the state where $\texttt{zz} = \texttt{0}$. We have $m(\emptyset) = 0$. It is easy to see that m is a morphism from $\langle S, c\rangle$ into $\langle T, d\rangle$. In fact m is a bijection and m^{-1} is a morphism, therefore m is an isomorphism.

As a consequence of the fact that m is a morphism, the executions of the operations enter and maximum from corresponding states will produce corresponding states and the same outputs. This give a precise mathematical sense to the sentences "R2 behaves like M1", "M1 and R2 cannot be distinguished by their outputs".

A specific feature of this example is that the substitutions in both machines are deterministic. We will mention briefly some issues.

In the following we will suppose that all proof obligations concerning machines and their refinements have been discharged.

Consider a machine MM with a state space S defining an operation op and its refinement RR with a state space T. The gluing invariant R_{GI} establishes a relation R between S and T. The difference between the notion of refinement and the notion of bismulation can be expressed as follows:
For a refinement:
Given a state $s, s \in S$ such that op can be executed from state s and a state $t, t \in T$ such that $sR_{GI}t$, there exists s' and t' such that $\texttt{op}_{\texttt{MM}}(s) = s'$, $\texttt{op}_{\texttt{RR}}(t) = t'$, and $s'R_{GI}t'$.
For a simulation relation R_S:
Given state s and s' such that MM can execute op for state s and $s' \in \texttt{op}_{\texttt{MM}}(\texttt{s})$ and a state t such that sR_St, there exists t' such that $t' \in \texttt{op}_{\texttt{RR}}(\texttt{t})$ and $s'R_St'$.

Therefore, if RR refines MM, then there exists a simulation relation between the state spaces $T' \subseteq T$ and S. The relation R_{GI} induced by the gluing invariant is contained in the simulation induced by the gluing invariant, with T' such that $T' = \texttt{dom}(\texttt{R}_{GI})$. The relaxation of the precondition for a refinement ensures that $S = \texttt{ran}(\texttt{R}_{GI})$. Intuitively, the relaxation of the precondition ensures that no state that should be taking into account by R_S. Operations returning values give the same results for corresponding states in S and T. We found that the gluing invariant plays a role analogous to a loop invariant in the Hoare proof system: When a developer writes a refinement of a machine he has the gluing invariant in his mind, in the same way as when he writes a loop, he has the loop invariant in his mind. Expliciting the invariant is one of the difficult points in the B methodology. The gluing invariant can be seen as indications (which can be a complete indication as in the example concerning machine `M1` and `R2`) about the simulation relation.

Finally let us say that we do not claim that one concept can take into account all aspects of the other, for example the question of the conservation of properties is not dealt with by coalgebra theory, but that there exists a lot of similarities that should be mentioned.

5 Conclusions

It is a banality to say that formal methods are indispensable. It is quite another matter to convince students of this fact. One of our goals is that students should become familiar with several usual formal modelling methods and their usage. It is with concrete examples of applications that students become aware of the use of such methods. Let us cite algebraic specification [4], first order logic and proof systems, model checking, CCS, Petri nets [5], Performance Evaluation Process Algebra (PEPA) [6]. All these models are taught with their applications, meaning that we integrate them with the courses dedicated to the studies of different systems and their associated software, when possible. These formalisms are used to treat different aspects of the modelled systems: feasibility, performance, correctness and observability, respectively. One of our goals is to show students that it is necessary to use these formal methods. The mathematics for observation of systems have reached a maturity level such that we find it necessary to introduce them in our curriculum for computer sciences. It is a natural complement to algebraic specifications, and we find that they help unify concepts for parallelism and distribution as well as concepts linked to refinement in B.

A question that still has to be answered in our school is at which point these mathematics should be taught. In an ideal teaching environment, it should be taught at the same time as algebraic specifications. Because of the time frame, this seems difficult, and a specific introduction to the subject can be given independently, before the course on B and on parallel systems.

A complete treatment of the relationships between coalgebra, bisimulation, and B machines, would be beyond the scope of this conference. We hope that we have given enough arguments to show that these relationships should be treated in a curriculum on software engineering.

References

[1] Jean-Raymond Abrial, *The B-Book: Assigning Programs to Meanings*, Cambridge University Press, 1966.

[2] Luca Aceto, Anna Ingòlfsdòttir, Kim Guldstrand Larsen, Jiri Srba, *Reactive Systems Modelling, Specification and Verification*, Cambridge University Press, 2007.

[3] Alexander, Cindy Kong, *Defining a Formal Coalgebraic Semantics for The Rosetta Specification Language*, Journal of Universal Computer Science, Vol. 9, no. 11 (2003), pp. 1322–1349.

[4] Marie-Claude Gaudel, Gille Bernot, Bruno Marre, Françoise Schlienger, *Précis de Génie Logicel*, Masson,1996.

[5] Annie Choquet-Geniet, *Les réseaux de Petri: Un outil de modélisation*, Dunod 2006.

[6] Jane Hillston. *A Compositional Approach to Performance Modelling*, Cambridge University Press, 1996.

[7] Bart Jacobs, *Draft: Introduction to Coalgebra. Towards Mathematics of States and Observations*, Institute for Computing and Information Sciences, Radboud University Nijmegen.

[8] Bart Jacobs, Jan Rutten, *A Tutorial on (Co)Algebras and (Co)Induction*, Bulletin of the European Association for Theoretical Computer Science, Vol. 62, pp. 222–259, 1997.

[9] Robin Milner, *A Calculus of Communicating Systems*, Springer Verlag, 1980.

[10] Horst Reichel, *Behavioural equivalence – a unifying concept for initial and final specifications.* Third Hugarian Computer Science Conference. Akademiai Kiado, Budapest, 1981.

[11] Davide Sangiogi, *On the origins of bisimulation, coinduction and fixed points*, Universita of Bologna, Italy, 2007.

[12] SEI, *Progress Maturity Profile*, March 2006.