Equivalence between executable OOZE and algebraic specification.

Vicent-Ramon Palasí Lallana.*

Barcelona, March 1995.

Abstract

In this paper two algorithms are presented: one which turns an executable OOZE specification into an algebraic specification and another which does the reverse operation. In this way, we shall be able to prove in a constructive way that executable OOZE and algebraic specification are equivalent and that, generally speaking, state-based specification is equivalent to to algebraic specification. We shall discuss whether these results can be made to cover other object-oriented Z dialects.

1 Introduction

Formal methods are a constant subject of research within the field of Software Engineering. They have been being developed since the mid 70's, and are nowadays beginning to be applied in industry (see [Hal90]) and are expected to become a most favourable tool for saving the time and effort which is spent in the development of applications.

This very same objective is shared by object-oriented programming. This new style of programming, which helps to reduce costs during the development process, has developed in a spectacular way over the past years and has now enter industry with great strength.

It is not strange, then, that over the last years there has been an effort to try to combine these two techniques, in order to make the most of the advantages that each of such techniques offers. The result has been a collection of different object-oriented formal methods. Although the number of methods is high (for a comprehensive view, see please [SBC92] and [LaH94]), they can be classified into two groups which correspond to two completely different approaches.

*I want to thank Allen Peralta for his interest and his comments on this paper and Pilar Nivela for her information on algebraic specification.

On the one hand, there has been an attempt to apply algebraic specification to object-orientation (as, for instance, in FOOPS and OBJ3). This type of specification is particularly suitable for the new paradigm. Indeed, algebraic specification is based on the notion of abstract data type (ADT, from now onwards) and, Meyer (in [Mey88]) already defined object-orientation as "the construction of software systems as structured collections of ADT implementations".

On the other hand, there has also been an attempt to take state-based specification (also called "model-based specification") as a starting-point for solving the problems posed. Even though the best known representative of this type of specification is Z, it cannot be applied to object-orientation, for it lacks the concept of class, which is fundamental. As a consequence, Z had to be extended, in order to be able to support classes. Thus, OOZE, MooZ, Z++, Object-Z, etc, emerged.

The intention of this paper is to produce constructive evidence which shall prove that these two approaches are equivalent to a certain extent. In other words, within the field of object-orientation, algebraic specification and state-based specification are equivalent. To be more precise, I shall be introducing two algorithms for converting executable OOZE into algebraic specification, thus proving that they are actually equivalent.

The structure of this paper is as follows. Firstly, I shall make a short description of the two methods under examination. Then, I shall explain the algorithms for conversion between both methods. Next, I shall draw conclusions and discuss whether these conclusions can or cannot be generalized to cover all the Z extensions for object-orientation.

2 Description of the examined models

2.1 Algebraic specification

The idea behind algebraic specification is extremely simple. It consists in expressing software behaviour via simple equations. For this reason, we shall take an approach which could be termed as "object-oriented". To be more specific, an algebraic specification is a set of ADT's. Each ADT is a type of data which is defined via their operations. Likewise, the operations of an ADT are defined via algebraic equations[1].This makes ADT syntax be quite similar to what follows:

[1]Depending on the type of algebraic specification, the nature of these equations may be quite different. In this paper, we will presume that we will use algebraic specifications with conditional equations. Therefore the equations will have the following form:

$$cond[a_1, ..., a_n] \Rightarrow op[a_1, ..., a_n] = exp[a_1, ..., a_n]$$

where op is the name of the operation defined by the equation

ADT name

    uses

        – ADT's whose operations are used by the present ADT–

    public

        – Names of the operations that can be used from other ADT's–

    signature

        – Names of the operations and their parameter and result types –

    equations

        – Equations which describe the behaviour of the operations –

end name

(Of course, the syntax which has been described, even though it can be regarded as being representative, is not the standard syntax in algebraic specification. Actually, there is no standard notation for describing algebraic specifications. However, the differences between the various notations tend to be syntactical rather than real).

As it can be noticed, I am making a distinction between two types of operations which can be defined in an ADT. On the one hand, there are public operations, which are those that can be used from other ADT's. On the other hand, there are hidden operations, which cannot be used fron the outside and which only serve as an aid for defining public operations.

An example of algebraic specification is given below (it is assumed that ADT's "String" and "Cardinal" have already been defined):

ADT Debtor

    uses

        String, Cardinal;

    public

        crdebtor, askname, askdebt;

    signature

        crdebtor: $\longrightarrow$ Debtor;

        askname: Debtor $\longrightarrow$ String;

        askdebt: Debtor $\longrightarrow$ Cardinal;

    equations

        $\forall$ n:String, d: Cardinal;

        askname(crdebtor(n,d)) $\equiv$ n;

        askdebt(crdebtor(n,d)) $\equiv$ d;

end Debtor.

ADT DebtorList –list of debtors–

uses

Debtor, String, Cardinal;

public

create, adddebtor, remdebtor, totaldebt

signature

create: $\longrightarrow$DebtorList;

adddebtor: Debtor $\times$ DebtorList $\longrightarrow$ DebtorList;

remdebtor: String $\times$ DebtorList$\longrightarrow$DebtorList

totaldebt: DebtorList$\longrightarrow$Cardinal

equations

$\forall$ d: Debtor, l:DebtorList, s:String;

remdebtor(s,create) $\equiv$ create

askname(d)=s $\Rightarrow$remdebtor(s,adddebtor(d,l)) $\equiv$ remdebtor(s,l);

askname(d) $\neq$ s $\Rightarrow$remdebtor(s,adddebtor(d,l)) $\equiv$ adddebtor(d,remdebtor(s,l));

totaldebt(create) $\equiv$ 0;

totaldebt(adddebtor(d,l))=askdebt(d)+totaldebt(l);

end DebtorList

As regards this ADT, I should like to comment on a few important points concerning algebraic specification. Firstly, the behaviour of the operations is described via equations which relate the operations among one another. In fact, in algebraic specification all the operations are defined in relation to a few (in this case, empty and push) which are called generating constructors and whose main feature is their abibity to generate all the values of the type in question.

Secondly, the equations described can be conditional or recursive (in fact, the last equation is recursive and conditional at the same time). The computation of an algebraic expression is done via rewriting, that is, by applying the equations of the specification to the expression until there is no equation which can be applied (then we can say that the expression is in normal or irreducible form).

As it can be seen, this approach we have taken is very similar to the one used for object-orientation. As a matter of fact, the only difference between an ADT and a class lies in the fact that, in the latter, methods are described in an imperative language, whereas, in the former, methods are described in terms of equations.

Of course, what I have just provided is only a short reminder of the most external aspects of algebraic specification. For an introduction to this type of specification, please see [EhM85].

2.2 Executable OOZE

Created by A. J. Alencar and J. A. Goguen within the framework of the Programming Research Group of the University of Oxford, OOZE (Object-Oriented Z Environment) is one of the extensions which have been carried out on Z in order to adapt Z to the new object-oriented paradigm. OOZE is able to support multiple inheritance, overloading and dynamic binding. Unlike other Z extensions (such as Object-Z), OOZE is truly object-oriented, for it contains the concept of "class". For a description of OOZE, please see [AlG91a] and also [LaH94].

In this paper, we shall not be examining OOZE thoroughly but only a subgroup of it. In particular, we shall examine the OOZE subgroup which can be executed, which is called executable OOZE. The choosing of executable OOZE among all the existing object-oriented notations is not a trivial choice. Executable OOZE contains what is minimally essential for an object-oriented notation to be considered as "state-based". Therefore, all the other object-oriented dialects in Z can be regarded as an extension of the concepts contained in executable OOZE. I shall later give further reasons for the choice of executable OOZE.

The basic notion in executable OOZE is that of class. Since executable OOZE is a specification language, a class in executable OOZE contains the specification of a class from the programme we wish to specify. In other words, the only difference between a class in executable OOZE and a class in other object-oriented programming language is that the former is oriented towards specification and the latter is oriented towards implementation.

Since executable OOZE is based on Z, its classes can be seen as collections of Z schemata having a certain structure. Specifically, the structure of a class in executable OOZE is as follows:

```
┌─ class name < ancestor names ──────────────────
│ constants
│ ┌─ State ──────────────────────────────────
│ │ class attributes
│ ├──────────
│ │ class invariant
│ └──────────────────────────────────────────
│ ┌─ Init ───────────────────────────────────
│ │ initial values
│ └──────────────────────────────────────────
│ methods
└────────────────────────────────────────────────
```

in which $class-name$ is the name of the class that is being defined and $ancestor-names$ are the names of the classes this class inherits from. The *constants* are the fixed values, which are the same for all the instances of the class. The *state* describes the attributes which define the state of the class. The *methods* operate on to this state. Among the methods, we can distinguish one which is named *Init* and contains the initial value of the state.

In executable OOZE, attributes and methods can be public (if we can access them from other classes) or hidden (if we cannot do so). We can tell one type from the other because in the latter the expression [*hidden*] is written beside their definition.

Methods in executable OOZE cannot be any S schema but must have an special form:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\ m\ }} \\
\Delta L \\
p_1 : T_1 \\
. \\
. \\
. \\
p_n : T_n \\
\hline
a_1' = e_1 \\
a_2' = e_2 \\
. \\
. \\
. \\
\mathit{if}\ \ P \\
\hline
\end{array}
$$

in which L is the list made up of a_1, a_2, . . . , which are the attributes that have been changed for method m. $p_1, ..., p_n$ are the parameters of m. Both P and $e_1, e_2,$ . . . are predicates about $p_1, p_2,$ . . . and the state of the class.

The intuitive sense of this construction is quite clear to all those who are familiar with Z. What this schema tells is that equations $a_1' = e_1$, $a_2' = e_2$, . . . must hold after method m has been executed. There is, though, a new element which is not present in Z: the if-clause.

This clause can be regarded as a precondition. It means that equations $a_1' = e_1$, $a_2' = e_2$, . . . only hold if predicate P is deemed true. This clause is optional: if it is not present the equations must still hold in any case. There can even be several if-clauses, each one of them being related to a set of conditional equations.

As it can be noticed, executable OOZE contains what is minimally required for a model to be considered as state-based. Like any state-based model, it contains a representation of the state of the class via attributes and methods which operate on to this state. In executable OOZE, the changes of state which are produced by the methods are described in an absolutely explicit way (via equations which indicate the atributes' values after the method has been executed). In other models, these values can be described implicitly, which makes them richer in terms of expressive power.

What I have just been doing is to provide a brief description of executable OOZE. For a more comprehensive and thorough description of executable OOZE and of OOZE in general, please see [AlG91b] and [AlG92].

I should like to conclude this section by providing an example of an specification which has been elaborated with the aid of executable OOZE. As in the previous case, I shall try to specify a list of debtors. Thus, we shall be able to contrast both styles of specification. The example is given below. (It is assumed that type STRING has already been defined).

Debtor

> *State*
> $name : STRING$
> $debt : \mathbb{N}$

DebtorList

> *State*
> $debtl : seq\ Debtor$

> *Init*
> $debtl' = \langle\,\rangle$

> *AddDebtor*
> $\Delta debtl$
> $x? : Debtor$
> ---
> $debtl' = debtl \frown \langle x? \rangle$

> *RemDebtor*
> $\Delta debtl$
> $s? : STRING$
> ---
> $debtl' = \langle\,\rangle$
> $if\ \ debtl = \langle\,\rangle$
> ---
> $debtl' = (tail\ \ debtl) \bullet RemDebtor(s?)$
> $if\ \ debtl \neq \langle\,\rangle \wedge (head\ \ debtl) \bullet name = s?$
> ---
> $debtl' = \langle head\ \ debtl \rangle \frown (tail\ \ debtl) \bullet RemDebtor(s?)$
> $if\ \ debtl \neq \langle\,\rangle \wedge (head\ \ debtl) \bullet name \neq s?$

continuation of DebtorList

TotalDebt

$sum! : \mathbb{N}$

$sum! = 0$
$if\ \ debtl = \langle\,\rangle$

$sum! = (head\ \ debtl) \bullet debt + (tail\ \ debtl) \bullet TotalDebt$
$if\ \ debtl \neq \langle\,\rangle$

As we shall see further on, an algebraic specification is based on its equations, whereas a specification in executable OOZE is based on a predefined type (in this case, that of sequence). This means that a specification in executable OOZE will require more predefined concepts and, besides, will be more oriented towards implementation than an algebraic specification.

3 Algorithms for conversion

In this chapter we shall examine two algorithms. The first one turns an executable OOZE specification into an equivalent algebraic specification. The second one does the reverse operation: taking an algebraic specification as a starting-point, it produces an equivalent executable OOZE specification.

3.1 The notion of equivalence

First of all, I should define exactly what I mean by the term "equivalent". In fact, in this case, the notion of equivalence is no trivial matter, for we are dealing with two absolutely different types of specification. On the one hand, model-based specification is based on the concept of state, which is a concept quite alien to algebraic specification. On the other hand, algebraic specification contains deductive properties (for example, it includes the possibility of arriving at inductive theorems) and it would be naive to try to translate them into executable OOZE. Therefore, what is it that we understand when we say that an OOZE specification and an algebraic specification bear an "equivalence"?

We shall consider two specifications as equivalent if they *behave* alike. In other words, two specifications are equivalent if they are *observationally* indistinguishable, that is to say, if they produce the same result when applied to the same object. Among the diverse operations, we can make a distinction between constructors (those operations which return a result of the type we are defining) and consultors (the result of which does not belong under the type we are defining). If this can be done, then we can say that two specifications are equivalent if their corresponding consultors produce the same result from the same combination of constructors (namely, the same object).

As it can be noticed, my approach is completely behavioral. As a matter of fact, this equivalence is almost the same as the "behaviorally equivalence" used in behavioral semantics. The only difference is that the latter is only applied to algebraic specifications, whereas the former can be applied to any type of object-oriented specification.

Let us be more precise. An specification's operations can be public or hidden. Obviously, we are not interested in hidden operations, since we have taken an observational approach. This leads us to the following definition.

Presupposing an specification S, we name call the set of its public operations $oper(S)$. Analogously, we shall name the set of its public constructors $construc(S)$ and the set of its public consultors $consul(S)$. We shall define $object(S)$ as the set of terms representing objects from the specification. In other words, $object(S)$ is the set of "ground terms" consisting of a combination of constructors from S. That is:

- m $\in object(S)$, if m is an initialling constructor.
- m$[a_1, ..., a_n] \in object(S)$, if m is a non-initialling constructor and $a_1, ..., a_n \in object(S)$.

(Let us remark that indicate the application of an operation on to its parameters by means of square brackets "[" and "]").

Assuming two algebraic specifications S_1 and S_2 we shall say that they are observationally equivalent (or simply equivalent) if there is a bijection β between $oper(S_1)$ and $oper(S_2)$, so that:

$$\forall\, m \in consul(S_1),\ t \in object(S_1), \quad m[t] = \beta(m)[\beta^*(t)]$$

where $\beta^*(t)$ is obtained from t, by replacing all the appearances of any operation o in t by $\beta(o)$. If S_1 and S_2 are observationally equivalent, we shall indicate it by means of the notation $S_1 \approx S_2$.

(Of course, this definition can be reversed: we could exchange the roles of S_1 and S_2 in the definition and this will be still true. In this case, the bijection would be the inverse of the previous bijection, that is to say, β^{-1}).

If one of the specifications should not be algebraic but model-based, then the definition would be as follows:

Assuming S_1 is an algebraic specification and S_2 is a model-based specification, then we shall say that they are observationally equivalent (or simply equivalent) if there is a bijection β between $oper(S_1)$ and $oper(S_2)$, so that:

$$\forall\, m \in consul(S_1),\ t \in object(S_1), \quad m[t] = \beta(m)[\sigma(t)]$$

where $\sigma(t)$ is defined in the following way:

If we apply the same t operations via the same t parameters to S_2, the resulting state will be called $\sigma(t)$. (Comment: notice that, strictly speaking, we are not dealing with the same operations but that for each operation o of t we apply $\beta(o)$). In fact, $\sigma(t)$ would be the closest notion to what would be "the translation of term t into OOZE". This is so because a translation in the true sense of the word would be impossible, since OOZE does not contain the concept of term, for its operations are applied in a dynamic way (and do not form a static term, as it is the case with algebraic specification).

3.2 Algorithm for conversion of executable OOZE into algebraic specification

This algorithm takes a specification P in OOZE as a starting-point and finds an algebraic specification A which is observationally equivalent to P.

This algorithm consists of several steps, which are explained next. In order to simplify the description, we have assumed that the type which defines the OOZE specification is referred to as T, and the public attributes are named at_1, at_2, ..., at_n, which are, respectively, of the type $t_1, t_2, ..., t_n$.

1. - Obtain the signature.

- **1. 1. - Signature of public operations**
  - The signature of $\beta(Init)$ is $\beta(Init) :\longrightarrow T;$.
  - For all the other methods m from the OOZE specification which are not "Init":
    - $\beta(m)$ has type T as a parameter.
    - if the definition of m contains a Δat_i, then $\beta(m)$ has type T as a result type.
    - if the definition of m contains a $i : t_{input}$, then $\beta(m)$ has type t_{input} as a parameter type.
    - if the definition of m contains a $o : t_{output}$, then $\beta(m)$ has type t_{output} as a result type.
- **1. 2. - Signature of hidden operations**

- The hidden operation cons is defined via signature cons: $t_1, ..., tn \longrightarrow$T.
- For each attribute at_i contain in the OOZE specification, we define, in a hidden way:
  - $getat_i$ via signature $getat_i : T \longrightarrow t_i$;
  - $setat_i$ via signature $setat_i : T \times t_i \longrightarrow T$.

2. - Obtain the equations.

- **2. 1. - Public operations**
  - We add equation $\beta(Init)[parameters] \equiv cons[EX_1, ..., EX_n]$ in which EX_i is:
    - e_i, if attribute at_i has been initialled in "Init" via the statement of equality $at_i' = e_i$.
    - $\perp$, if attribute at_i has not been initialled in "Init".
  - For all the constructors m which are not "Init", we shall add equation $\beta(m)[parameters] \equiv cons[EX_1, ..., EX_n]$ where EX_i is:
    - e_i^*, if attribute at_i is modified in m via the statement of equality $at_i' = e_i$. (e_i^* is obtained from e_i, by replacing all the appearances of any attribute at_i by $getat_i(t)$, where t is the T type parameter which appears at the left-hand side of the equation being defined).
    - $getat_i(t)$, if attribute at_i is not modified in m.
  - For all consultors m we shall add equation $\beta(m)[parameters] \equiv e_i^*$ where e_i is the expression which returns consultor, and e_i^* is obtained from e_i in the same way as it was done for constructors.
  - If method m should happen to have preconditions $(P_1, P_2, ..., P_n)$, then for each set of conditional equations affected by a precondition P_i, we shall apply the process previously explained and we shall add a condition $P_i \Rightarrow$. We shall also add an equation $\neg P_1 \wedge \neg P_2 \wedge ... \wedge \neg P_n \Rightarrow m[parametres] \equiv$ error.

- **2. 2. - Hidden operations**
  - From each i from 1 to n, we shall define the following equations:
    - $E_i \neq \perp \Rightarrow getat_i[cons[E_1, ..., Ei, ..., E_n]] \equiv Ei$;
    - $getat_i[cons[E_1, ..., \perp, ..., E_n]] \equiv$error;
    - $setat_i[cons[E_1, ..., Ei, ..., E_n], E_{new}] \equiv cons[E_1, ..., E_{new}, ...En]$;

3. - Translate the invariant

- If the class in executable OOZE has no invariant, then the equivalent algebraic specification would be the one obtained in steps 1 and 2.

- However, if the class has an invariant "I", it is necessary to modify the set of equations which has been obtained as follows:

  Any equation $C \Rightarrow m[parameters] \equiv expression$ which has been obtained (in which m is a public constructor), is turned into three equations:

  mm[parameters]≡expression
  $I^* \wedge C$ ⇒m[parameters] ≡expression
  ¬ I^*⇒m[parameters]≡error

  where mm is a hidden operation, and I^* is obtained from invariant I by replacing any reference to any attribute at_i by $getat_i(mm[parameters])$.

Explanatory notes on this algorithm:

- This algorithm presupposes that Z basic types (numerical types, sets, sequences, . . .) are already defined (through their operations) in the algebraic specification.

- It also presupposes that all the expressions in executable OOZE style are translated into algebraic specification style. As a matter of fact, in OOZE the way of indicating an operation is via $object \bullet operator(other-parameters)$, whereas in algebraic specification the way of indicating an operation is via $operator(object, other-parameters)$. It is assumed that the algorithm makes this (easy) change.

- The demonstration of this algorithm is to be found in Annex B.

- As it can be seen, the idea underlying this algorithm is that which considers that each attribute is used by the methods in two different ways: either with a view to consulting it or with a view to updating it. In this manner, we are able to give shape to an attribute by means of two operations: that through which we consult it ($getat_i$) and that through which we update it ($setat_i$). Thus, any reference to the attribute can be replaced by an operation and, in this way, we can define the state of an object by means of equations. Taking this as a basis, we shall be able to turn any state-based specification into an specification based on equations.

- We can also demonstrate that any "ground term" from the algebraic specification we have obtained has a normal form $cons(a_1, ..., a_n)$ where a_i is of the type t_i . In other words, when we translate an OOZE specification into an algebraic one, the state of object O in OOZE (which is the value of its attributes) is translated as the normal form of the "ground term" corresponding to O.

3.3 Algorithm for conversion of algebraic specification into executable OOZE.

The algorithm for constructing an specification S in executable OOZE which is equivalent to a given algebraic specification consists of the following steps:

1. - Change from algebraic specification into canonical form

The first thing to do is to put the algebraic specification into a certain form (which we shall call "canonical form"), from which the shift to executable OOZE is particularly easy. In order to do so, we shall follow the procedures which are explained below.

We shall call T the type which defines the algebraic specification. Any equation belonging to this specification will have the following form:

$$cond[a_1, ..., a_n] \Rightarrow op[a_1, ..., a_n] \equiv exp[a_1, ..., a_n]$$

Thus, for each equation belonging to the algebraic specification, we shall:

- Translate the expressions which act as parameters of *op* into the condition, replacing them in *op* by simple variables:

  $$v_1 = a_1 \wedge ... \wedge v_n = a_n \wedge cond[a_1, ..., a_n] \Rightarrow op[v_1, ..., v_n] \equiv exp[a_1, ..., a_n]$$

  in which v_i is a variable of the same type as a_i.

- Apply the following procedures to $exp[a_1, ..., a_n]$ and $cond[a_1, ..., a_n]$:
  - We shall replace any appearance of a_i by v_i.
  - Once we finish doing that, we shall replace any appearance of parameter number k belonging to a_i by the expression $param(v_i, k)$.
  - Afterwards, we replace any appearance of subparameter number j belonging to parameter number k belonging to a_i by the expression $param(param(v_i, k), j)$.
  - We shall repeat this procedure until all subexpressions from a_i have been replaced both in *cond* and in *exp*.

- For each symbol of any operation $v_i = a_i$,
  - If a_i has no parameters, we shall leave it as it is.
  - If a_i is of the form $operac[p_1, ..., p_n]$, we shall replace $v_i = a_i$ by $header(v_i) = operac \wedge param(v_i, 1) = p_1 \wedge ... \wedge param(v_i, n) = p_n$. Then we apply recursively this procedure on to $param(v_i, 1) = p_1, ..., param(v_i, n) = p_n$.

[COMMENT: *header* is the operation which obtains the operator "head" of an expression -namely, the first one- and *param* is the operation which obtains the parameter number n of an expression. They could be defined in the following way:

$arg \neq op(p_1, ..., p_n) \Rightarrow header(arg) \equiv error$
$header(op(p_1, ..., p_n)) \equiv op$
$arg \neq op(p_1, ..., p_n) \Rightarrow param(arg, n) \equiv error$
$param(op(p_1, ..., p_n), 1) \equiv p_1$
$\vdots$
$param(op(p_1, ..., p_n), n) \equiv p_n$]

2. - Change from canonical form into executable OOZE

Each ADT from the algebraic specification is turned into an OOZE class, including its methods.

- The resulting state of the OOZE class is as follows:

State
$st : TERM;$

  in which the meaning of TERM is explained further on (in section headed "Explanatory notes on this algorithm").

- Each method belonging to the OOZE class corresponds to an operation of the algebraic specification. In particular,

  – For each "OP" operation which has, in canonical form, a set of equations such as the following:

  $$COND_1[v_1, v_2, ..., v_n] \Rightarrow OP[v_1, v_2, ..., v_n] \equiv EXP_1[v_1, v_2, ..., v_n]$$
  $$\vdots$$
  $$COND_M[v_1, v_2, ..., v_n] \Rightarrow OP[v_1, v_2, ..., v_n] \equiv EXP_M[v_1, v_2, ..., v_n]$$

  in which $v_1, v_2, ..., v_n$ are variables (since we are dealing with canonical form). The type of v_1 is T and the types of $v_2, ..., v_n$ are, respectively, $t_2, ..., t_n$. We shall call TR the type which returns the operation "OP", that is, the type of $OP[v_1, v_2, ..., v_n]$ and $EXP_M[v_1, v_2, ..., v_n]$. (If OP is a constructor, TR is T).

  – The resulting method in executable OOZE is:

* If OP is a consultor:

$$
\begin{array}{|l}
\hline
\beta(OP) \\
v_2? : t_2 \\
\vdots \\
v_n? : t_n \\
r! : TR \\
\hline
r! = \beta^*(EXP_1[st, v_2?, ..., v_n?]) \\
\mathit{if}\ \beta^*(COND_1[st, v_2?, ..., v_n?]) \\
\hline
\vdots \\
\hline
r! = \beta^*(EXP_M[st, v_2?, ..., v_n?]) \\
\mathit{if}\ \beta^*(COND_M[st, v_2?, ..., v_n?]) \\
\hline
\end{array}
$$

* If OP is a constructor:

$$
\begin{array}{|l}
\hline
\beta(OP) \\
\Delta st \\
v_2? : t_2 \\
\vdots \\
v_n? : t_n \\
\hline
st' = \beta^*(EXP_1[st, v_2?, ..., v_n?]) \\
\mathit{if}\ \beta^*(COND_1[st, v_2?, ..., v_n?]) \\
\hline
. \\
. \\
. \\
\hline
st' = \beta^*(EXP_M[st, v_2?, ..., v_n?]) \\
\mathit{if}\ \beta^*(COND_M[st, v_2?, ..., v_n?]) \\
\hline
st' = apply(``\beta(OP)'', \langle st, \beta^*(v_2?), ..., \beta^*(v_n?)\rangle) \\
\mathit{if}\ \neg\ \beta^*(COND_1[st, v_2?, ..., v_n?]) \wedge \\
\vdots \\
\wedge\ \neg\ \beta^*(COND_M[st, v_2?, ..., v_n?]) \\
\hline
\end{array}
$$

in which "$\beta(OP)$" is the name of the operation (in the form of a string of characters), $\beta^*(expression)$ is obtained by replacing all the appearances of an operation o in *expression* by $\beta(o)$, and the meaning of apply is explained below (in section headed "Explanatory notes on this algorithm").

Explanatory notes on this algorithm

- This algorithm presupposes that the class TERM has already been defined. TERM is a class which implements the structure of "ground terms" of the algebraic specification in executable OOZE. It consists of three operations: *apply*, through which we construct a term from n subterms and an operator which is applied to those terms. For a given term, *header* will return the operator heading it. For a given term, $param(k)$ will return its own subterm number k. A possible specification of TERM can be found in Annex A.

- It is also assumed that all the expressions are translated in object-oriented notation style. Indeed, in OOZE the way of indicating an operation is via $object \bullet operator(other - parameters)$, whereas in algebraic specification the way of indicating an operation is via $operator(object, other - parameters)$. Despite this, for the description of the previous algorithm we have kept in both cases the algebraic style of notation (for simplicity's sake).

- The demonstration of this algorithm can be found in Annex C.

- As it can be noticed, the idea underlying this algorithm is quite simple. We take as a basis the notion that what performs the role of the state in algebraic specification is the structure of the instances' terms (this was made clear in the last explanatory note on the previous algorithm). Thus, we shall translate the algebraic specification into a class whose state is the structure of terms of such specification. Bearing this idea in mind, we shall then proceed to translate all algebraic operations into OOZE methods.

4 Conclusions and future research.

We have examined two algorithms which allow the change from algebraic specification into executable OOZE and viceversa. Thus, we have prove, in a constructive manner, that these two specification systems are equivalent. In other words, we have seen that specifying classes by taking the concept of state as a basis is equivalent to specifying them via equations. As a matter of fact, we have proved that the role the state plays in model-based specification is the same role the structure of terms performs in algebraic specification. In this respect, what in the former is a dynamic "object"[2] in the latter is modelled syntactically (and, therefore, statically).

An interesting question is whether this conclusion we have reached can be made to cover all object-oriented dialects in Z. The answer is no. In fact, the expressive power of object-oriented dialects in Z tends to be greater than that in algebraic specification. However, this does not mean that algebraic and model-based specifications are not equivalent.

[2]Here the meaning of object is the classic one used in Computer Science, and not the one used in object-orientation

I shall explain this point next.

As we have seen, a model-based specification specifies each class taking as a basis the notion of state. Class methods are defined from the changes they produce in this state. Z, however, is not only a state-based model but, in addition, other expressive possibilities are incorporated in it (for instance, it is possible to define the result of a method in an implicit way). These expressive possibilities are the ones that cannot be translated into algebraic specification. Nevertheless, these espressive possibilities are not really a part of the concept of "state-based specification". For instance, a model such as executable OOZE is "state-based", even though these possibilities are not incorporated into it.

On the other hand, the algorithms presented in this paper, apart from their theorical importance as constructive evidence of the equivalence between the two styles of specification, have practical application. For example, taking an specification in executable OOZE as a starting-point, we have deductive possibilities available, which seemed impossible. We only need to turn it into an equivalent algebraic specification. Likewise, if we have a system which works out the specification through OOZE (as the one that is being elaborated by our colleague Allen Peralta [Per++]), we shall be able to incorporate into it all our work already conducted on algebraic specification (in an absolutely automatic way). With this aim in mind, we are then in a position to implement the algorithm described in this paper.

Moreover, this paper is part of the thesis entitled "Automatic Verification of Object-Oriented Programming", in which we aim to build an automatic verifier for programs specified algebraically. With the conversion algorithms described in this paper , the research can also be extended to executable OOZE. This will provide OOZE-users with a method for proving the correctness of programs, which will also mean a cost-reduction for the software industry.

5 References

[AlG91a] ALENCAR, A. J GOGUEN, J. A *OOZE: An Object Oriented Z Environment.* ECOOP '91 Proceedings, vol. 512, 180-199.

[AlG91b] ALENCAR, A. J GOGUEN, J. A *Two Examples in OOZE* Technical Report TR-25-91, Programming Research Group, University of Oxford.

[AlG92] ALENCAR, A. J GOGUEN, J. A *OOZE with examples* Technical Report TR-7-92, Programming Research Group, University of Oxford.

[EhM85] EHRIG, H. MAHR, B. *Fundamentals of Algebraic Specification.* ETACS monographs on theoretical computer science, Springer-Verlag, Berlin, 1985.

[Hal90] HALL, A. *Seven Myths on Formal Methods.* IEEE Software, september 1990, 11-19.

[LaH94] LANO, K. HAUGHTON, H. *Object-Oriented Specification Case Studies.* The Object-Oriented Series, Prentice-Hall, Redwood Books, 1994.

[Mey88] MEYER, B. *Object-Oriented Software Construction.* C. A. R. Hoare Series Editor, Prentice-Hall, University Press, Cambridge, 1988.

[Per++] PERALTA, A. *Eiffel++: Un Llenguatge per al Desenvolupament Orientat a Objectes de Sistemes d'Informació.* Master thesis, directed by Pere Botella i López, to appear.

[SBC92] STEPNEY, S. BARDEN, R. COOPER, D. *Object Orientation in Z.* Workshops in Computing, Springer-Verlag, 1992.

A A possible specification of the class TERM

A possible specification of the class TERM is produced below. In other to make it more readable, I have given the name *emptynode* to the Init of NODE and the name *apply* to the Init of TERM (in this way they can be distinguished more easily).

The most suitable data structure to model TERM is an n-ary tree. However, in OOZE only lineal structures are predefined and, consequently, the specification of TERM is rendered a little bit more complex.

SNAT

State

$snat : seq\ \mathbb{N}$

---

addnsn

$number? : \mathbb{N}$

$r! : snat$

$r! = \langle\,\rangle$

$if\ \ snat = \langle\,\rangle$

$r! = ((snat \bullet head) + number?) \frown ((snat \bullet tail) \bullet addnsn(number?))$

$if\ \ snat \neq \langle\,\rangle$

(*to be continued on next page*)

continuation of SNAT

contsn

$n? : \mathbb{N}$
$r! : \mathbb{N}$

$r! = head\ snat$
$if\ \ n? = 1$

$r! = (snat \bullet tail) \bullet contsn(n? - 1)$
$if\ \ n? > 1$

NODE

State

$oper : OPERATOR$
$children : snat$

emptynode

$op? : OPERATOR$

$oper' = op?$
$children' = \langle\,\rangle$

addnnode

$number? : \mathbb{N}$
$r! : node$

$r! \bullet oper = oper$
$r! \bullet children = children \bullet addnsn(number?)$

TERM

State

$term : seq\ node$

(to be continued on next page)

continuation of TERM

apply

$op? : OPERATOR$
$seqt? : sterm$

$term' = \langle emptynode(op?) \rangle$
$if\ \ seqt? = \langle\,\rangle$

$term' = (seqt? \bullet embapply(0)) \frown (seqt? \bullet finalnode(op?))$
$if\ \ seqt? \neq \langle\,\rangle$

param

$n? : \mathbb{N}$
$result! : term$

$result! = (term \bullet paramless(nodparam)) \frown ((term \bullet contterm(nodparam)))$

header

$result! : OPERATOR$

$result! = (term \bullet last) \bullet oper$

addsnumber

$number? : \mathbb{N}$
$r! : term$

$r! = \langle\,\rangle$
$if\ \ term = \langle\,\rangle$

$r! = ((term \bullet head) \bullet addnnode(number?)) \frown ((term \bullet tail) \bullet addsnumber(number?))$
$if\ \ term \neq \langle\,\rangle$

paramless

$n? : \mathbb{N}$
$result! : term$

$result = \langle\,\rangle$
$if\ \ term = \langle\,\rangle$

$result = term \bullet tail$
$if\ \ term \neq \langle\,\rangle \wedge n? = 1$

$result = (term \bullet head) \frown ((term \bullet tail) \bullet paramless(n? - 1))$
$if\ \ term \neq \langle\,\rangle \wedge n > 1$

(to be continued on next page)

continuation of TERM

contterm

$n? : \mathbb{N}$
$r! : node$

$r! = term \bullet head$
$if\ \ n? = 1$

$r! = (term \bullet tail) \bullet contterm(n? - 1)$
$if\ \ n? > 1$

nodparam

$n? : \mathbb{N}$
$result! : node$

$result! = (term \bullet last \bullet children) \bullet contsn(n?)$
$if\ \ term \neq emptyseq \wedge$
$\#((term \bullet last) \bullet children) \leqq n?$

STERM

State

$sterm : seq\ term$

embapply

$number? : \mathbb{N}$
$tr! : term$

$tr! = \langle\,\rangle$
$if\ \ sterm = \langle\,\rangle$

$tr! = ((sterm \bullet head) \bullet addsnumb(number?))$
$\quad ^\frown((sterm \bullet tail) \bullet embapply(number? + \#(sterm \bullet head)))$
$if\ \ sterm \neq \langle\,\rangle$

finalnode

$op? : OPERATOR$
$res! : node$

$res! \bullet oper = op?$
$res! \bullet children = sterm \bullet thefirsts(1)$

(*to be continued on next page*)

$$
\begin{array}{|l}
\hline
\textit{continuation of STERM} \\
\begin{array}{|l}
\hline
\textit{thefirsts} \\
num? : \mathbb{N} \\
res! : snat \\
\hline
res! = \langle\, \rangle \\
\mathit{if}\ \ sterm = \langle\, \rangle \\
\hline
res! = num? \frown ((sterm \bullet tail) \bullet thefirsts(num? + \#(sterm \bullet head))) \\
\mathit{if}\ \ sterm \neq \langle\, \rangle \\
\hline
\end{array} \\
\hline
\end{array}
$$

B Demonstration of the first algorithm

We intend to prove that the algebraic specification produced by the first algorithm discussed in this paper is observationally equivalent to class in executable OOZE.

The definition of observational equivalence is applied in this case as follows:

Let us assume that S_1 is a specification in executable OOZE and S_2 is an algebraic specification. We shall say that they are observationally equivalent is there is a bijection β between $oper(S_1)$ and $oper(S_2)$, whereby:

For each state s of the first one which has been obtained by applying a combination of constructors c to S_1 the following statement is fulfilled:

$$\forall\, m \in consul(S_1), \quad m[s] = \beta(m)[k(s)]$$

where $k(s)$ is obtained via the combination of constructors c, by replacing any appearance of a constructor *cons* by $\beta(cons)$.

(It can be noticed that this definition is equivalent to that which appears in section 3. 1. In fact, the definition is obtained by inverting it).

In the case of this algorithm, taking an OOZE specification S as the starting-point, we find a $\beta(S)$, which is not only equivalent to the OOZE specification but also β is the bijection of equivalence. Therefore, in order to prove the correctness of this algorithm, we must prove:

$$\forall\, m \in consul(S), \quad m[s] = \beta(m)[k(s)]$$

In order to do so, we shall demonstrate the following:

Lemma For each s defined as above, and for each attribute at_i

$s(at_i) = getat_i(k(s))$

(where $s(at_i)$, means "the value of attribute at_i in the state s").

We shall prove this by structural induction on the combination of constructors s. (For a definition of structural induction, see [EhM85]).

- **Structural induction base**. c has no parameters and, so, it is an initialling constructor, that is to say, *Init*. We shall distinguish between two cases:
  - *Init* initials attribute at_i via the statement of equality $at'_i = e_i$. In this case, $s(at_i)$ is obviously e_i. According to the algorithm, $\beta(Init) \equiv cons(EX_1, ..., EX_i, ...EX_n)$ where $EX_i = e_i$ and $getat_i(cons(EX_1, ...EX_i, ..., EX_n)) \equiv EX_i$. By applying these equations, we obtain $getat_i(\beta(Init)) \equiv e_i$ and, therefore, $getat_i(\beta(Init)) \equiv s(at_i)$.
  - *Init* does not initial attribute at_i. In this case, $s(at_i)$ would return error. On the other hand, in this case, $\beta(Init)$ is defined as $cons(EX_1, ..., \perp, ..., EX_n)$ and $getat_i(cons(EX_1, ..., \perp, ..., EX_n))$ returns error, whereby $getat_i(\beta(Init)) \equiv s(at_i)$.
- **Structural induction step** We shall assume that c is of the form $operac(p_1, ..., p_n)$. We shall distinguish between two cases:
  - operac modifies attribute at_i via the statement of equality $at'_i = e_i$. In this case, $s(at_i)$ is obviously $s(e_i)$[3]. According to the algorithm, $\beta(operac(p_1, ..., p_n)) \equiv cons(EX_1, ..., e^*_i, ...EX_n)$ where e^*_i was defined in section 3. 2. Thus, $getat_i(cons(EX_1, ..., e^*_i, ...EX_n)) \equiv e^*_i$

    The only difference between e^*_i and $s(e_i)$ is : in the place where in the former there is $getat_i(t)$[4], in the latter there is $s'(at_i)$ (where s' is the state corresponding to t, that is, t=k(s')). However, since t is shorter than c, we can apply the hypothesis of induction, $getat_i(t) = s(at_i)$. Therefore, $e^*_i = s(e_i)$ and, consequently, it is proved that $s(at_i) = getat_i(k(s))$.
  - operac does not modify at_i. In this case, $s(at_i) = s'(at_i)$, for the previous value is kept. However, by definition of algorithm, $\beta(operac(p_1, ..., p_n)) \equiv cons(EX_1, ..., getat_i(t), ...EX_n)$. Thus, $getat_i(\beta(operac(p_1, ..., p_n))) \equiv getat_i(t)$.

[3] Like $s(at_i)$, $s(e_i)$ is defined as the value of expression e_i in the state s. In other words, $s(e_i)$ is e_i via substitution of any appearance of at_i by $s(at_i)$

[4] t was defined in section 3. 2

Since t is shorter than c, via the hypothesis of induction, $s'(at_i) = getat_i(t)$ and, consequently, we have proved our point.

Now we are in a position to prove the theorem:

Theorem S_1 i S_2 are observationally equivalent or, in other terms:

$$\forall\, m \in consul(S), \quad m[s] = \beta(m)[k(s)].$$

(using the notation given at the beginning of this section)

Proof:

As we already know, m is a consultor and the result which it returns is defined by means of an expression e_i. Thus, $m[s] = s(e_i)$- by use of the definition of $s(e_i)$ given in footnote 2. On the other hand, $\beta(m)$ has been defined in the form of $\beta(m)[k(s)] = e_i^*$ (where this term is the one defined before).

As we have earlier seen, the only difference between e_i^* and $s(e_i)$ is: in the place where in the former there is $getat_i$, in the latter there is $s(at_i)$. However, by use of the lemma proved before, these two expressions are equivalent. Therefore, the theorem has been proved.

Note on the demonstration

The translation of the preconditions and the invariant has not been demonstrated, since it is evident to anyone familiar with these concepts and it is not my intention to try the reader's patience. (The preconditions and invariant are conditions which, when not fulfilled, return error. Therefore, it is not worth translating them as conditional equations of error).

C Demonstration of the second algorithm

In this appendix I shall prove the correctness of the algorithm for converting algebraic specification into executable OOZE. I shall divide this demonstration into two parts.

C.1 Demonstration of the change from algebraic specification into canonical form.

We intend to prove that the change from algebraic specification into canonical form is correct, that is to say, that the specification in canonical form which is obtained is equivalent to the original specification.

I shall first introduce the notation. Presupposing an algebraic specification S, this algorithm of change into canonical form produces a new specification which is $\Phi(S)$. We want to prove that, for any S:

$$S \equiv \Phi(S) \tag{1}$$

Since this algorithm consists of three different steps, we could divide the change that is applied to the algebraic specification into three different changes: Φ_1, Φ_2 and Φ_3, so that $\Phi = \Phi_3 \circ \Phi_2 \circ \Phi_1$.

If we can prove that $\Phi_1(S) \equiv S$, $\Phi_2(S) \equiv S$ and $\Phi_3(S) \equiv S$, we shall them prove (1), since:

$$\Phi(S) \equiv (\Phi_3 \circ \Phi_2 \circ \Phi_1)(S) \equiv \Phi_3(\Phi_2(\Phi_1(S))) \equiv \Phi_3(\Phi_2(S)) \equiv \Phi_3(S) \equiv S$$

$\Phi_1(S) \equiv S$ i $\Phi_2(S) \equiv S$ is trivial (see the algorithm).

We shall prove that $\Phi_3(S) \equiv S$. In fact, what we must prove is that, for any $v_i = a_i$, $\Phi_3(v_i = a_i) \equiv (v_i = ai)$. We shall do so via structural induction[5]:

- **Structural induction base.** If i has no parameters, $\Phi_3(v_i = a_i) = (v_i = ai)$ by the definition of Φ_3 in this case. Thus, trivially, $\Phi_3(v_i = a_i) \equiv (v_i = ai)$.

- **Structural induction step.** The expression $v_i = a_i$, means that v_i is equal to i. That is to say, that both the header v_1 and all of its parameters are the same as of a_i. This could be put in proper form in this way:

  $$(v_i = a_i) \equiv (header(v_i) = header(a_i)) \land param(v_i, 1) = param(a_i, 1) \land \newline ... \land param(v_i, n) = param(a_i, n))$$

  As, in this case, a_i is of the form $operac(p_1, ..., p_n)$:

  $$(v_i = a_i) \equiv (header(v_i) = header(operac(p_1, ..., p_n))) \land \newline \land\, param(v_i, 1) = param(operac(p_1, ..., p_n), 1) \land \newline ... \land param(v_i, n) = param(operac(p_1, ..., p_n), n))$$

  Via the definitions of header and param, this expression of equality is changed into:

  $$(v_i = a_i) \equiv (header(v_i) = operac) \land param(v_i, 1) = p_1 \land ... \land param(v_i, n) = p_n$$

  And, via the hypothesis of induction, we obtain:

[5] Actually, there is a misuse of language in applying Φ_3 to $v_i = a_i$, as this is an expression and Φ_3 has only been defined over specifications. It be necessary to define another operation ϕ_3 over expressions. I have choosen to leave it as it is so as not to make the notation excessively complicated.

$$(v_i = a_i) \equiv (header(v_i) = operac) \wedge \Phi_3(param(v_i, 1) = p_1) \wedge ... \wedge \Phi_3(param(v_i, n) = p_n)$$

On the other hand, by the defintion of Φ_3 in this case:

$$\Phi_3(v_i = a_i) \equiv (header(v_i) = operac) \wedge \Phi_3(param(v_i, 1) = p_1) \wedge \\ ... \wedge \Phi_3(param(v_i, n) = p_n)$$

Therefore, $\Phi_3(v_i = a_i) \equiv (v_i = a_i)$, and so it is proved that $\Phi(v_i = a_i) \equiv (v_i = a_i)$, which was what intended to prove.

C.2 Demonstration of the change from canonical form into executable OOZE

We intend to prove that the change from the algebraic specification in canonical form into executable OOZE is correct.

First of all, I shall introduce some notation. Presupposing an algebraic specification S, we shall refer to the specification in OOZE which is obtained via the algorithm as $\beta(S)$. Analogously, for any type T from the algebraic specification, its corresponding class in OOZE will be called $\beta(T)$. For any algebraic operation o, we shall refer to the corresponding operation in OOZE as v$\beta(o)$[6]. Analogously, for any given term g, we shall denote by $\beta^*(g)$ the term which is obtained by replacing all the operations o in g by $\beta(o)$.

Presupposing t is a term from the algebraic specification, if we apply to $\beta(T)$ the same operations of t via the same parameters of t, the resulting state of $\beta(T)$ will be called $\sigma(t)$. (Comment: notice that, rigorously, we are not dealing with the same operations but that for each operation o we apply $\beta(o)$). Notice also that, strictly speaking, $\sigma(t)$ would the closest concept to that of "translation of term t" in OOZE. In fact, a translation in the true sense of the word would be impossible to accomplish, since OOZE cannot contain the concept of term, for its operations are dynamic (and not static, as it is the case with algebraic specification).

We shall use the following lemma:

Lema. For any $t \in object(S)$, where S is convergent, $\sigma(t)$ contains $\beta^*(normal(t))$.

We shall prove this via structural induction:

- **Structural induction base** Since t has no parameters, it is an initialling constructor. Thence, $\beta^*(t) = \beta(t)$ and $normal(t) = t$, by the definitions of β^* and $normal$. In addition for the definition we shall make of method $\beta(t)$ (see the algorithm), it

[6] I have preferred to give the same name to very similar concepts so as not to render the notation more complex.

follows that $\sigma(t) = \beta(t)$. By applying the above equations, we obtain the following: $(\sigma(t) = \beta(t)) \equiv (\sigma(t) = \beta(normal(t))) \equiv (\sigma(t) = \beta^*(normal(t)))$.

- **Structural induction step** Let us asssume t is of the form $oper(p_1, ..., p_n)$. Now we want to verify that $\sigma(oper(p_1, ..., p_n))$ contains $\beta^*(normal(oper(p_1, ..., p_n)))$.

  From the definition of σ, it follows that $\sigma(oper(p_1, ..., p_n))$ is equal to $\beta(oper)(\sigma(p_1), ..., \sigma(p_n))$. Via the hypothesis of induction:

$$\sigma(oper(p_1, ..., p_n)) = \beta(oper)(\beta^*(normal(p_1)), ..., \beta^*(normal(p_n))). \quad (2)$$

  On the other hand, in the algebraic specification S, the following statement of equality is fullfilled: $normal(oper(p_1, ..., p_n)) = oper(normal(p_1), ..., normal(p_n))$, since it is a convergent specification and the order of the reductions is not important. Then, by applying β^* to the two sides of this equation:

$$\beta^*(normal(oper(p_1, ..., p_n))) = \beta^*(oper(normal(p_1)), ..., normal(p_n)).$$

  And, since, by the definition of β^*,

$$\beta^*(oper(normal(p_1)), ..., normal(p_n)) = \\ \beta(oper)(\beta^*(normal(p_1)), ..., \beta^*(normal(p_n)))$$

  Finally, the result is:

$$\beta^*(normal(oper(p_1, ..., p_n))) = \beta(oper)(\beta^*(normal(p_1)), ..., \beta^*(normal(p_n))) \quad (3)$$

  By applying (2) i (3), we obtain:

$$\sigma(oper(p_1, ..., p_n)) = \beta^*(normal(op(p_1, ..., p_n)))$$

  which was what we intended to prove.

Theorem Pressuposing a convergent algebraic specification S, then S and $\beta(S)$ are observationally equivalent (that is, S $\approx \beta(S)$) .

Proof:

By the definition of observacional equivalence, the theorem can be expressed as follows:

$$\forall\, m \in consul(S),\ t \in object(S), \quad m[t] = \beta(m)[\sigma(t)]$$

Since S is convergent, m(t)=m(normal(t)) and, via the lemma earlier proved, $\sigma(t) = \beta^*(normal(t))$, whereby we obtain:

$$m(normal(t)) = \beta(m)[\beta^*(normal(t))]$$

which is true by the definition of $\beta(m)$ in this algorithm. (Since we have defined $\beta(m)$ in such a way that the only difference between m and $\beta(m)$ is that the latter deals with terms translated via the bijection β).

Notes on this demonstration:

- As it can be seen, this demonstration only deals with the case in which S is convergent. It can be proved that, if S is not convergent because it does not have the property of "halting", the OOZE specification which is obtained by means of this algorithm is still equivalent to the algebraic specification, but there is no assurance that the execution will be completed. If S is not convergent because it does not have the property of confluence, the algorithm will return an OOZE class equivalent to the confluent algebraic specification which is obtained by imposing the order of application "inside-out" to the original non-confluent specification.
- In order to facilitate the demonstration, it has been obviated that OOZE does not follow the same system of notation as the one used in algebraic specification. As a matter of fact, in OOZE the way of representing an operation is object • operator(other -parameters), whereas in algebraic specification the way of representing an operation is operator(object, other - parameters). For the demonstration I have used (for the sake of convenience and uniformity) the algebraic notation style for OOZE.