FORMAL SPECIFICATION AND REFINEMENT FOR AN INTERACTIVE WEB EXAMPLE

Ingrid van Coppenhagen
School of Computing, University of South Africa (Florida Campus), Private Bag X6, Florida, South Africa, 1710

Barry Dwolatzky
School of Electrical & Information Engineering, University of the Witwatersrand, Private Bag 3, Wits, South Africa, 2050

Keywords: Implementation, interactive, refinement, specification, Web, XML, Z.

Abstract: This paper provides a small interactive Web example (the Car1 example) that illustrates parts of the software life cycle processes of specification, refinement and implementation in an object-oriented environment. Part of the software system is specified in Z, data- and operation refined and then implemented into HTML, XML, XSD and JavaScript. Short descriptions of the refinement processes comprising data refinement, operation refinement and operation decomposition are given. The main focuses of the study are to firstly investigate how effective (or not) a formal specification is for an interactive Web system, and secondly to illustrate a selection control structure in the refinement process.

1 INTRODUCTION

This paper evaluates the specification of a part of a small interactive Web system (the Car1 example) in Z, the subsequent data and operation refinement, and then the implementation into HTML, XML, XSD and JavaScript.

One of the aims of the paper is to evaluate how effective a formal specification is for the Car1 example, with particular emphasis on the use of a selection control structure. This control structure has been specified and refined formally and in detail.

In sections 2, 3, and 4 the concepts of specification, refinement, and control structures are described respectively. From Section 5 the Car1 example is presented, with the previous three concepts included. In Section 9 some conclusions are drawn. Following Section 9 are the references, Appendix A which contains the programs of the Car1 example and Appendix B which explains some Z notation.

2 SPECIFICATION

The Z notation uses mathematical concepts, particularly set theory, to specify data and operations. This allows for reasoning about systems, for example checking the consistency of the data and the various operations, as well as verifying the correctness of subsequent system development during refinement (Ratcliff, 1994, Lightfoot, 2001, Smith, 2000, Woodcock, 1996, Jacky, 1997).

3 REFINEMENT

The two main stages of refinement are data refinement and operation refinement. Data and operation refinement can be looked at as that part of the development process that corresponds to the design phase of the traditional software life cycle. Ways to represent the abstract data structures that will be more amenable to computer processing are chosen, as well as the translation of abstract operations into corresponding concrete operations. The concrete operations are, however, still expressed in the language of schemas (using Z) and describe only the relationship among the components of before and after stages. This does not indicate how

van Coppenhagen I. and Dwolatzky B. (2006).
FORMAL SPECIFICATION AND REFINEMENT FOR AN INTERACTIVE WEB EXAMPLE.
In *Proceedings of WEBIST 2006 - Second International Conference on Web Information Systems and Technologies - Internet Technology / Web Interface and Applications*, pages 89-96
DOI: 10.5220/0001237100890096

such changes of state are to be expressed in an implementation computer language (Ratcliff, 1994, Jacky, 1997, Derrick, 2001).

In operation refinement the process of conversions of descriptions of state changes can be carried into executable instruction sequences.

4 CONTROL STRUCTURES

All programs can be written in terms of only three control structures, namely the *sequence structure,* the *selection structure* and the *repetition structure.* The **if** structure is called a *single-selection structure,* because it selects or ignores a single action. The **if/else** structure is called a *double-selection structure,* because it selects between two different actions (or groups of actions) (Deitel, 2002).

5 THE CAR1 EXAMPLE

In brief, in the Car1 example, the information of 6 cars is given in an XML document car1.xml. This information is used to display the Web site demonstrated in Figure 2. The make, model number, year, price and picture of each car are given. The price of some cars is more than R200000.00 and some are less. The user can choose either one to get a display of the make, model number, year and price for the relevant cars. This *if* selection structure in the JavaScript program constitutes the crux of the discussion regarding the refinement of the specifications of the programs.

6 Z SPECIFICATIONS

Basic types, sets, data, constants, choices and messages: Basic types (given sets):
[*MAKE, MODEL, YEAR, PRICE, IMAGE, X*]

Refer to figures 1. The following are these sets:
$X = \{01, 02, 03, 04, 05, 06\} \in \mathbb{N}$ This represents the record counter for the data.
$MAKE = \{(01, \text{Peugeot}), (02, \text{Audi}), (03, \text{Citroën}), (04, \text{Mazda}), (05, \text{Mercedes}), (06, \text{Mercedes})\} \in X \rightarrow\!\!\!\!\!\!\rightarrow MAKE$
$MODEL = \{(01, 307CC), (02, A6), (03, C5), (04, 1.6), (05, \text{E-class}), (06, \text{SLRMcLaren})\} \in X \rightarrow\!\!\!\!\!\!\rightarrow MODEL$
$YEAR = \{(01, 2004), (02, 2004), (03, 2004), (04, 2004), (05, 2002), (06, 2004)\} \in X \rightarrow\!\!\!\!\!\!\rightarrow YEAR$
$PRICE = \{(01, 250000.00), (02, 340000.00), (03, 150000.00), (04, 170000.00), (05, 540000.00), (06, 640000.00)\} \in X \rightarrow\!\!\!\!\!\!\rightarrow PRICE$
$IMAGE = \{(01, \text{car(01).jpg}), (02, \text{car(02).jpg}), (03, \text{car(03).jpg}), (04, \text{car(04).jpg}), (05, \text{car(05).jpg}), (06, \text{car(06).jpg})\} \in X \rightarrow\!\!\!\!\!\!\rightarrow IMAGE$
STR::= *Price*| *Year*| *Model*| *Over*| *Under*; $\mathbb{N}$: Natural numbers; $\varnothing$: Empty set; *maxSize*: $\mathbb{N}$; *n*: $\mathbb{N}$ The number of cars. For this example n = 6.

Table 1: Car1 data file (given in car1.xml).

X (record counter)	make	model	year	img	price
01	Peugeot	307CC	2004	car(01).jpg	250000.0
02	Audi	A6	2004	car(02).jpg	340000.0
03	Citroën	C5	2004	car(03).jpg	150000.0
04	Mazda	1.6	2004	car(04).jpg	170000.0
05	Mercedes	E-class	2002	car(05).jpg	540000.0
06	Mercedes	SLRMcLaren	2004	car(06).jpg	640000.0

Car_System

*car*1.*html* (User Interface)
*car*1.*js*; *car*1.*xml*; *car*1.*xsd*

Figure 1a: The Z Car system.

*car*1.*html*

Display Welcome page; On click: *car*1.*js* [Detail of this specification not relevant to the discussion]

Figure 1b: The Car1.html file (User Interface).

*car*1.*xml*

$x: \mathbb{P}\, X$ [*X* is not a field in the record, it is used only in the schema as a record counter]
make: $X \rightarrow\!\!\!\!\!\!\rightarrow MAKE$; *model*: $X \rightarrow\!\!\!\!\!\!\rightarrow MODEL$;
year: $X \rightarrow\!\!\!\!\!\!\rightarrow YEAR$; *img*: $X \rightarrow\!\!\!\!\!\!\rightarrow IMAGE$;
price: $X \rightarrow\!\!\!\!\!\!\rightarrow PRICE$; *n*: $\mathbb{N}$; *car*1.*xsd*;

(*x* = dom *make*; dom *model*; dom *year*; dom *img*; dom *price*); *n* = 6

Figure 1c: car1.xml.

*car*1.*xsd*

Ξ *car*1.*xml*
[Information not relevant to the discussion of this paper]

Figure 1d: car1.xsd.

car1.js

Ξ *car1.html*; Ξ *car1.xml*; Ξ *car1.xsd*
make!; *model*!; *price*!; *img*!; *year*!
n: ℕ; *n* = 6; *str*?: *STR*

Init
x′ = ∅

showPix()
(∀*i*:0..(*n*-1)• *make*(*i*)! ∧ *model*(*i*)! ∧ *img*(*i*)!)

range(*str*)
(*str*? = '*Over*') ⇒ (∃ *i*:0..(*n*-1) • *price*(*i*) > 200000 ∧ *make*(*i*)! ∧ *model*(*i*)! ∧ *price*(*i*)! ∧ *year*(*i*)! ∧ *img*(*i*)!)
(*str*? = '*Under*') ⇒ (∃ *i*:0..(*n*-1) • *price*(*i*) < 200000 ∧ *make*(*i*)! ∧ *model*(*i*)! ∧ *price*(*i*)! ∧ *year*(*i*)! ∧ *img*(*i*)!)

ListAll(*str*)
[Information not relevant to the discussion of this paper]

Figure 1e: car1.js.

7 VERIFICATION AND REFINEMENT (Z)

The Z data specifications for the schema *Car1.xml* data types and operation specifications for the schema *range*(*str*) can be refined as follows:

7.1 Verifying Consistency of Global Definitions

For the axiomatic description

GlobalDeclarations

GlobalPredicates

it must be established that there exists values for *GlobalDeclarations* that satisfy *GlobalPredicates*. For example from the type definitions, the state variables can be defined as follows:

x: ℙ*X*; *make*: *X* ⇸ *MAKE*; *model*: *X* ⇸ *MODEL*

(*x* = dom *make*; dom *model*)

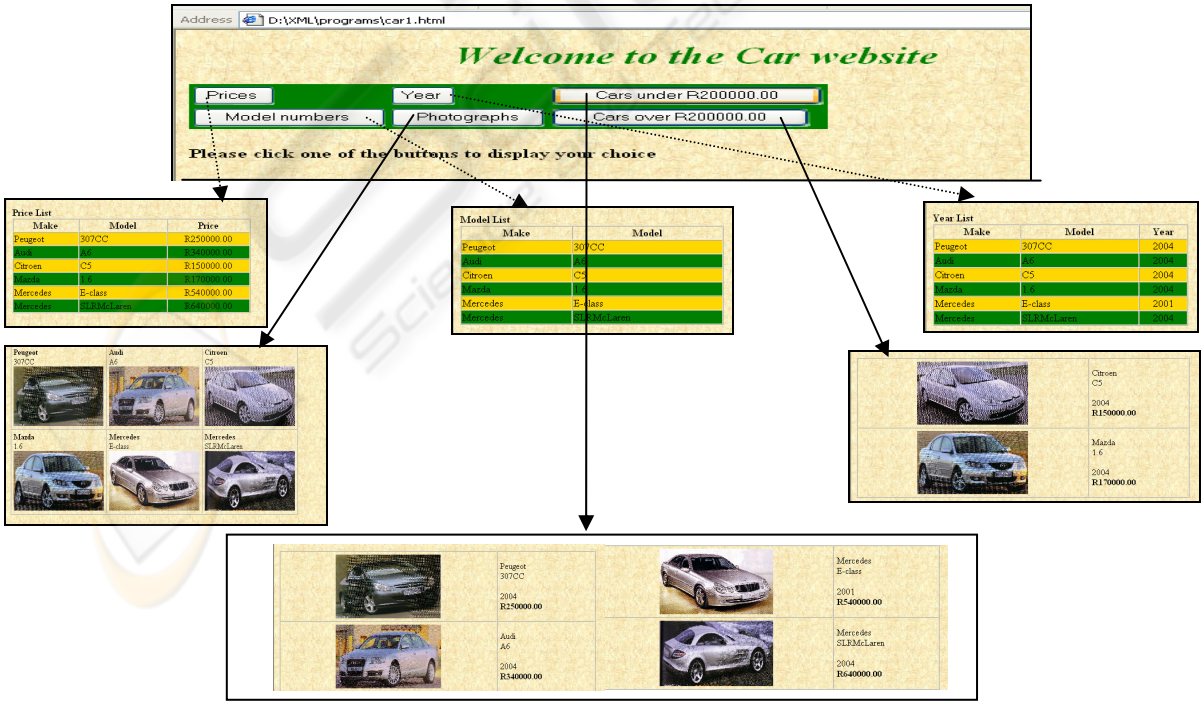

Figure 2: The Car Website.

Suppose x = {03} and *make* = {Citroën}. Therefore x = dom *make*. Therefore $\vdash \exists x: \mathbb{P}X \wedge \exists make: X \rightarrow\!\!\!\!\!\!+\; MAKE \bullet true$. By a simple property of logic $\vdash$ *true*. Also

make: *MAKE* (declaration of constant)
make = Citoën (value of constant)

This description is consistent because it does not contradict *x*′s declaration.

A number of consistency checks can be performed on the state space of Figure 1c.

x: *X*; *make*: *MAKE*
x = 03; *make* = Citroën

This axiomatic description is also a verification of the consistency of the global definitions or $\vdash \exists Car1.xml' \bullet x' \in X \wedge make' \in MAKE$ which for the axiomatic description is true, therefore $\vdash \exists Car1.xml' \bullet true$ and by a simple property of logic $\vdash$ *true*. This satisfies the verification for global definitions, which can also serve as part of a data refinement because a concrete data type was constructed that simulates the abstract one.

7.2 Verifying Consistency of an Initial State

A check must be done to ensure that a consistent initial state exists. This check can be expressed as the *initialisation theorem* which has the following general form: $\vdash \exists State' \bullet InitStateSchema$ which can be extended to $\vdash \exists State'; Inputs? \bullet InitStateSchema$ if there are input variables for the initial state schema *InitStateSchema*.

The concrete initial state for *Car*1.*js* is:

init
$x' = \varnothing$

To show that it is consistent:
$\vdash \exists x': \mathbb{P}X; \exists make': X \rightarrow\!\!\!\!\!\!+\; MAKE \bullet x' =$ dom *make*′.

If dom *make*′ = ∅ then *x*′ = ∅ which implies that there is a state *Car*1.*js*′ of the state definition schema that satisfies the initial state description.

The above mentioned checks can also serve as data refinement where it must be determined whether every abstract state has at least one concrete representative and there exists a consistent initial state.

7.3 Verifying Consistency of Operations

For an operation that is defined as: OperationDeclarations | OperationPredicates, the consistency theorem is $\vdash \exists$ *OperationDeclarations* • *OperationPredicates*.

Calculating its precondition can check an operation's consistency. If the operation is inconsistent, its precondition will be *false.* A false precondition strongly suggests a defect in the operation description (Ratcliff, 1994).

The consistency theorem for the operation *range*(*str*) will be: $(str? = \text{`}Over\text{'}) \Rightarrow (\exists i:0..(n\text{-}1) \bullet price(i) > 200000 \wedge make(i)! \wedge model(i)! \wedge price(i)! \wedge year(i)! \wedge img(i)!)$.

Now assume that $(str? = \text{'}Over\text{'}) \Rightarrow false$ then $(\exists i:0..(n\text{-}1) \bullet price(i) < 200000 \wedge make(i)! \wedge model(i)! \wedge price(i)! \wedge year(i)! \wedge img(i)!)$.

But according to the second statement in *range*(*str*) (Figure 1e): if $(str? = \text{`}Under\text{'}) \Rightarrow (\exists i:0..(n\text{-}1) \bullet price(i) < 200000 \wedge make(i)! \wedge model(i)! \wedge price(i)! \wedge year(i)! \wedge img(i)!)$.

Then according to a simple property of logic $\vdash (str? = \text{`}Over\text{'}) \Rightarrow (\exists i:0..(n\text{-}1) \bullet price(i) < 200000 \wedge make(i)! \wedge model(i)! \wedge price(i)! \wedge year(i)! \wedge img(i)!) \bullet false$ and then according to a simple property of logic $\vdash$ *false.* This means that the assumption was *false,* and that the sequent predicate is not a contradiction, hence that *range*(*str*) is consistent.

This means that the assumption was *false*, and that the sequent predicate is not a contradiction, hence that *range*(*str*) is consistent.

7.4 Data Refinement

Refer to Figure 1c, the abstract state *Car*1.*xml,* with an initial state:

*init_Car*1.*xml*
*Car*1.*xml*′
$x' = \varnothing$

and with (for example) an abstract operation on the data being the operation schema *range*(*str*) (from Figure 1e).

We plan to implement this system into a programming language that supports arrays and lists. We decide to refine the abstract specification to a detailed design based on sequences because we expect this will be easier to map into the target programming language. The reason for this is because sequences are sorted lists.

For the refinement the concrete representation of *Car*1.*xml* is not a set but a sequence *Car*2.*xml* of elements of type X, $X \rightarrow\!\!\!\!\!\!\rightarrow MAKE$, $X \rightarrow\!\!\!\!\!\!\rightarrow MODEL$, $X \rightarrow\!\!\!\!\!\!\rightarrow YEAR$, $X \rightarrow\!\!\!\!\!\!\rightarrow IMAGE$ and $X \rightarrow\!\!\!\!\!\!\rightarrow PRICE$.

The concrete representative x is not a set but a sequence $x1$. The state set *Car*1.*xml* is re-expressed as:

*Car*2.*xml*

$x1$: seq X; *make*1: seq ($X \rightarrow\!\!\!\!\!\!\rightarrow MAKE$);
*model*1: seq ($X \rightarrow\!\!\!\!\!\!\rightarrow MODEL$); *year*1: seq ($X \rightarrow\!\!\!\!\!\!\rightarrow YEAR$);
*img*1: seq ($X \rightarrow\!\!\!\!\!\!\rightarrow IMAGE$); *price*1: seq ($X \rightarrow\!\!\!\!\!\!\rightarrow PRICE$);

(# ran $x1$ = # ran *make*1; # ran *model*1; # ran *year*1; # ran *img*1; # ran *price*1)

Here is the concrete selection control structure operation of *range*(*str*):

*range*1(*str*)

(*str*? = '*Over*') ⇒ (∃ *i*:0..(*n*-1) • *price*1(*i*) > 200000 ∧ *make*1(*i*)! ∧ *model*1(*i*)! ∧ *price*1(*i*)! ∧ *year*1(*i*)! ∧ *img*1(*i*)!)
(*str*? = '*Under*') ⇒ (∃ *i*:0..(*n*-1) • *price*1(*i*) < 200000 ∧ *make*1(*i*)! ∧ *model*1(*i*)! ∧ *price*1(*i*)! ∧ *year*1(*i*)! ∧ *img*1(*i*)!)

The sequence should always hold the same elements as the set. A sequence is a function from natural numbers to elements, so the elements stored in the sequence are the range of this function. The range of the sequence must be the same as the set.

x = ran $x1$ ∧ x' = ran $x1'$ also *price* = ran *price*1 ∧ *price*′ = ran *price*1′, etc..therefore (∃ *i*:0..(*n*-1) • *price*(*i*) > 200000 ⇔ (∃ *i*:0..(*n*-1) • ran *price*1(*i*) > 200000).

This must be true before and after any operation, so equations appear for unprimed and primed (′) variables. We now form the implication that expresses the refinement. The predicate of the abstract operation *range*(*str*) appears on the right of the implication arrow, and the predicate of the concrete operation *range*1(*str*) is on the left, along with the equations relating *make,* *make*1, *model*, *model*1, *year*, *year*1, *img*, *img*1, *price* and *price*1.

When (*str*? = '*Over*′) then (∃ *i*:0..(*n*-1) • *price*1(*i*) > 200000) ∧ *price* = ran *price*1 ∧ *price*′ = ran *price*1′. Also when (*str*? = '*Under*′) then (∃ *i*:0..(*n*-1) • *price*1(*i*) < 200000) ∧ *price* = ran *price*1 ∧ *price*′ = ran *price*1′

7.5 Proof of the Selection Control Structure Refinement

When (*str*? = '*Over*') ⇒ then (∃*i*:0..(*n*-1) • *price*1′(*i*) > 200000) = (∃*i*:0..(*n*-1) • *price*1(*i*) > 200000) ∧ *price* = ran *price*1 ∧ *price*′ = ran *price*1′ ⇒ *price*′ = *price*.
⇔ (∃*i*:0..(*n*-1) • *price*′(*i*) > 200000)
[Assume antecedent]
⇔ (∃*i*:0..(*n*-1) • ran *price*1′(*i*) > 200000)
[Antecedent *price*' = ran *price*1']
⇔ (∃*i*:0..(*n*-1) • ran *price*1(*i*) > 200000) [Given]
⇔ (∃*i*:0..(*n*-1) • *price*(*i*) > 200000)
[Antecedent *price* = ran *price*1]
⇔ *true*

An equivalent proof for when (*str*? = '*Under*′).

7.6 Verifying the Correctness of the Concrete Initial State

The concrete initial state must not describe initial states that have no counterpart in the abstract model (Ratcliff, 1994, Jacky, 1997, Derrick, 2001). A theorem of the following form is to be proved: Given the retrieve relation then: *InitConcState* ⊢ *InitAbsState* which says that 'for each concrete initial state, there is a corresponding abstract one'.

Refer to the following schema definitions:

*init_Car*1.*xml*

*Car*1.*xml*′

$x' = \varnothing$

and from the Data refinement the *Car*2.*xml* schema and

*Car*2.*xml′*

*x*1': seq *X*; *make*1': seq (*X* ⇸ *MAKE*);
*model*1' : seq (*X* ⇸ *MODEL*); *year*1': seq (*X* ⇸ *YEAR*);
*img*1': seq (*X* ⇸ *IMAGE*); *price*1': seq (*X* ⇸ *PRICE*);

(# ran *x*1' = # ran *make*1'; # ran *model*1'; # ran *year*1';
ran *img*1'; # ran *price*1')

and

*init_ Car*2.*xml*

*Car*2.*xml′*

*x*1′ = ⟨⟩

and from the Data refinement the *Car*1.*xml* schema and

CARel

*Car*1.*xml*; *Car*2.*xml*

x = ran *x*1

and

CARel′

*Car*1.*xml′*; *Car*2.*xml′*

x′ = ran *x*1'

It must be proved that there is a state *CARel′* of the general model *CARel* (concrete to abstract relation) that satisfies the following: *init_ Car*1.*xml* ⊢ *init_ Car*2.*xml*.

CARel′ acts as an extra hypotheses (given *CARel′*). The declarative part of the right-hand side schema text is just *Car*1.*xml′* which is provided by *CARel′* on the left. The sequent is then unfolded into *CARel′*; *Car*2.*xml′* | *x*1′ = ⟨⟩ ⊢ *x′* = ∅ which holds because *x*1′ = ⟨⟩ on the left and *x′* = ran *x*1′ in *CARel′*.

By substitution *x′* = ran ⟨⟩, and *x′* = ∅ immediately follows.

7.7 The concrete State Must be Consistent

It has to be shown in general that

⊢ ∃ *ConcState′* • *InitConcState* (*InitConcState* represents the initial concrete state) or for our example: ⊢ ∃ *Car*3.*xml′* • *init_Car*2.*xml*

*Car*3.*xml′* is a state of the general model *Car*3.*xml*.

From the Data refinement it is concluded that the state sets *x*: ℙ*X*; *make*: *X* ⇸ *MAKE*; *model*: *X* ⇸ *MODEL*; *year*: *X* ⇸ *YEAR*; *img*: *X* ⇸ *IMAGE*; *price*: *X* ⇸ *PRICE*; are implemented as arrays with an index variable: *x*1: **array** [0..(*maxSize*-1)]; *make*1: **array**[0..(*maxSize*-1)]; *model*1: **array** [0..(*maxSize*-1)]; *year*1: array[0..(*maxSize* -1)]; *img*1: **array**[0..(*maxSize*-1)]; *price*1: **array**[0..(*maxSize*-1)]; *n*: 0..(*maxSize*-1).

It is assumed that the *n* elements of the *x*1 array are sorted in ascending sequence to ensure that no duplicates are kept in the array and to facilitate fast lookup of the array. For all the *n* elements of the *x*1 array, the corresponding elements in the *make*1, *model*1, *year*1, *img*1, and *price*1 arrays have the same element number as the number of the *x*1 array. For example, *make*1(3) is the car make of the car represented by *x*1(3).

Add the following to *Car*1.*xml*: | *maxSize*: ℕ to give:

*car*3.*xml*

x: ℙ *X*; *make*: *X*⇸ *MAKE*; *model*: *X*⇸ *MODEL*;
year: *X*⇸ *YEAR*; *img*: *X*⇸ *IMAGE*;
price: *X*⇸ *PRICE*; *n*: ℕ; *car*1.*xsd*

(*x* = dom *make*; dom *model*; dom *year*; dom *img*; dom *price*); #*x*⩽ *maxSize* -1; *n* = 6

*Car*4.*xml*

*x*1: seq *X*; *make*1: seq (*X* ⇸ *MAKE*)
*model*1 : seq (*X* ⇸ *MODEL*); *year*1: seq (*X* ⇸ *YEAR*);
*img*1: seq (*X* ⇸ *IMAGE*); *price*1: seq (*X* ⇸ *PRICE*);
n = 0..(*maxSize* -1)

(# ran *x*1 = # ran *make*1; # ran *model*1; # ran *year*1;
#ran *img*1; # ran *price*1); #*x*1 = *n*
∀*i*,*j*: dom *x*1 • *i* < *j* ⇒ *x*1(*i*) < *x*1(*j*); *n*' = *n*

*Car*4.*xml*'

*x*1': seq *X*; *make*1': seq (*X* ⇸ *MAKE*);
*model*1' : seq (*X* ⇸ *MODEL*); *year*1': seq (*X* ⇸ *YEAR*);
*img*1': seq (*X* ⇸ *IMAGE*); *price*1': seq (*X* ⇸ *PRICE*);
n = 0..(maxSize -1)

(# ran *x*1' = # ran *make*1'; # ran *model*1'; # ran *year*1';
ran *img*1'; # ran *price*1'); #*x*1' = *n″*
∀*i*,*j*: dom *x*1' • *i* < *j* ⇒ *x*1'(*i*) < *x*1'(*j*); *n*' = *n*

To show that it is consistent: Refer to *Car*4.*xml*': ⊢ ∃ *x*1′: seq *X*; *n*′: 0..(*maxSize*-1); ∀*i, j*: dom *x*1′ • *i* < *j* ⇒ *x*1′(0) < *x*1′(1) < *x*1′(2), < .. *x*1′(*maxSize*-1);

$\#x1' = n'$. If $n' = 0$ then $x1' = \langle\rangle$ that implies that there is a state (*Car*4.*xml'*) of the general model *Car*4.*xml* that satisfies the initial state description *init_ Car*2.*xml*.

7.8 Determine Whether Every Abstract State Has at Least One Concrete Representative

This can be achieved by determining if each abstract variable can be derived or 'retrieved' from the concrete variables by writing down equalities of the form: *AbsVar* = *Expr*(*ConcVars*) where *AbsVar* represents an abstract variable of the abstract state, *Expr* an expression and *ConcVars* the concrete variable of the concrete state representing the abstract state.

For the example the predicate x = ran $x1$ will be referred to as the 'retrieve relation' *CARel* (concrete-to-abstract relation) that brings together the abstract and the concrete states:

CARel

*Car*1.*xml*; *Car*2.*xml*

---

x = ran $x1$

The equality means that *CARel* is in effect a total function when viewed as 'calculating' the abstract state from the concrete one. Being total means that every concrete state maps to some abstract state. This implicit property of the retrieve relation being functional and total, characterises the fact that a simplified form of data refinement is discussed (Ratcliff, 1994).

Suppose, however, the 'sorted' invariant was removed from *Car*2.*xml* so that the array element order was immaterial. Assume that no duplicates are stored in the array. The design will now include some redundancy in that each non-empty, non-singleton set in the abstract state would have more than one concrete representation (Ratcliff, 1994).

For example, the abstract state

⟨*model* ⇻ {307CC, A6, C5, 1.6, E-class, SLRMcLaren}⟩ will have 6! concrete representatives (of which two are shown): ⟨*model*1 ⇻ ⟨1.6, 307CC,.., C5⟩, *n* ⇻ 6⟩ and ⟨*model*1 ⇻ ⟨C5, 1.6,..A6⟩, *n* ⇻ 6⟩

In general, assuming no duplicates, there would be *n*! concrete representatives for a single abstract state. The implicit functionality of a retrieve relation such as *CARel* is not compromised because the relation expresses a calculation from *concrete to abstract* (Ratcliff, 1994).

7.9 Operation Refinement

Refer to *range*1(*str*) from the Data refinement with *n*' and *n* added:

*range*1(*str*)

*range*1(*str*)

n' = *n*

This is a data-refined operation because the abstract *range*(*str*) operation is re-expressed as the concrete operation *range*1(*str*). *range*1(*str*) is refined into *range*2(*str*):

*range*2(*str*)

(*str*? = '*Over*') ⇒ ((∃ *i*:0..(*n*-1) • *price*1(*i*) > 200000) ∧
(θ*Car*4.*xml'* = θ*Car*4.*xml*) ∧((*i*)↾ *make*1)! ∧ ((*i*)↾ *model*1)! ∧ ((*i*)↾ *price*1)! ∧ ((*i*)↾ *year*1)! ∧ ((*i*)↾ *img*1)!)
(*str*? = '*Under*') ⇒ ((∃ *i*:0..(*n*-1) • *price*1(*i*) < 200000) ∧
(θ*Car*4.*xml'* = θ*Car*4.*xml*) ∧((*i*)↾ *make*1)! ∧
((*i*)↾ *model*1)! ∧ ((*i*)↾ *price*1)! ∧ ((*i*)↾ *year*1)! ∧ ((*i*)↾ *img*1)!)
n' = *n*

((i) ↾ *make*1) extracts the ith element of array *make*1.

*range*2(*str*) can be refined into the following algorithm: (Let *n* be the number of Cars).

[lookFor(*str?, range*(*str*))*]*
[Send *str* to the function *range*(*str*) (in car1.js)]
for *i* = 0..(*n*-1)
if *str* = '*Over*' **and** *price*(*i*) > 200000.00
[Check for car prices > 200000]
display *make*(*i*), *model*(*i*), *year*(*i*), *img*(*i*), *price*(*i*)**;**
[Display the values on the website]
else
if *str* = '*Under*' **and** *price*(*i*) < 200000.00
[Check for car prices < 200000]
display *make*(*i*), *model*(*i*), *year*(*i*), *img*(*i*), *price*(*i*)**;**
[Display the values on the website]
endif
endif
endFor

This algorithm is implemented by the range(str) function of the car1.js program (refer to Appendix A).

8 IMPLEMENTATION

The programs are provided in Appendix A. The programs include a car1.html program that provides the user interface, the car1.xml and its corresponding schema program car1.xsd provide the data input, and car1.js is a javascript program that does the calculations. This aids in the understanding of the process of refinement if the end product (the programs) are given to indicate where the algorithms of the refinement are leading to. The output of the programs should correspond with the results of the data refinements and the instantiations. This serves a dual purpose in that the output can be verified against the instantiations, and the instantiations against the output. Therefore the implementation serves to verify that the specifications are correct, and the specifications serve to verify that the implementation is correct (Dong, 2004, Sun, 2002, Woodcock, 1996, Deitel, 2002, Doke, 2002, McGrath, 2002).

9 CONCLUSION

From the extensive refinement of the selection control structure used on the data file, it can be concluded that such a detailed specification and refinement as illustrated in this paper will definitely reduce errors in the coding of the programs. The application of formal specifications and refinements also serves a dual purpose in that the code can be verified against the specifications, and the specifications can be verified against the code.

REFERENCES

Ratcliff, B., 1994. *Introducing Specification using Z.* McGraw-Hill Book Company.

Dong, J.S., 2004. *Semantic Web and Formal Specifications.* Short version of ICSE'04 tutorial. Computer Science Department, National University of Singapore.

Sun, J., Song Dong, J., Liu, J., Wang, H., 2002. *A Formal Approach to the Design of ZML.* Annals of Software Engineering 13, 329-356.

Lightfoot, D., 2001. *Formal Specifications using Z.* Palgrave.

Smith, G., 2000. *The Object-Z specification language.* Kluwer Academic Publishers.

Woodcock, J., and Davies, J., 1996. *Using Z Specification, Refinement and Proof.* Prentice Hall.

Deitel, H. M., Deitel, P. J., and Nieto, T. R., 2002. *Internet & World Wide Web How to Program.* Prentice Hall.

Jacky, J., 1997. *The Way of Z.* Cambridge University Press.

Derrick, J., and Borten, E., 2001. *Refinement in Z and Z. Foundations and Advanced Applications,* Springer-Verlag London.

Doke, E., Satzinger, J. W., and Williams, S. Rebstock, 2002. *Object-Oriented Application using Java.* Course Technology, Thomson Learning.

McGrath, M., 2002. *XML in easy steps.* Computer Step.

APPENDIX A

http://www.unisa.ac.za/contents/colleges/col_science_eng_tech/docs/ingrid/AppendixA.doc

APPENDIX B

Z NOTATION

http://www.unisa.ac.za/contents/colleges/col_science_eng_tech/docs/ingrid/AppendixB.doc