Quantified Assertions in Eiffel

Stuart Kent and Ian Maung[*]
Department of Computing, University of Brighton,
Brighton BN2 4GJ, UK.

Email: Stuart.Kent@brighton.ac.uk, Ian.Maung@dcs.warwick.ac.uk

Abstract

This paper discusses extensions to the language Eiffel, required to write more comprehensive software specifications, where a specification in Eiffel is a collection of class interfaces with features specified using an assertion language (i.e. a BON static model). The focus of the paper is the extension of the assertion language with quantification. Two forms of quantification are identified, which are distinguished according to whether the quantified variable is of reference or expanded type. A semantics for each of the two forms is described, and the consequences for assertion checking at run-time considered.

1 Introduction

This paper considers the extension of the object-oriented (OO) programming/design language Eiffel with quantified assertions. The assertion language in Eiffel gives it some formal specification capability, in its support for contracts and seamless software development. It enables contracts to be specified precisely, extracted from the class definition source code automatically and monitored at run-time. Run-time checking assists in the location of bugs during testing and forms the basis of disciplined exception handling for the delivered system. It also checks that the implementation is consistent with the contract, particularly important during system maintenance.

As explained in (Waldén and Nerson 1994), Eiffel supports a truly seamless approach to software specification, design and implementation. Systems are specified using deferred classes giving the interfaces of problem domain classes (names, signatures and contracts of exported features), and the inheritance and clientship relationships between classes. During design, new solution domain classes are introduced (probably reused from a library) and secret and selectively exported features of the problem domain classes and their contracts are defined. Implementation introduces effective classes, implementing the deferred class interfaces and introducing feature bodies for deferred features. As software development progresses, no translation is required; progress is achieved simply by adding more detail until the system becomes executable. Most importantly, the same semantic model and the same abstraction mechanism (the class) is used throughout. This is in contrast to other OO approaches (e.g. (Cook and Daniels 1994; Booch 1994)) where different semantic models are used at each stage, requiring a potentially tedious, ill-defined and error-prone translation between the different models. Formal approaches to software development (e.g. (Jones 1990)) either suffer from the same problem or achieve seamlessness by using a toy programming language with inadequate abstraction constructs and facilities. Finally, as also noted in (Waldén and Nerson 1994), Eiffel/BON supports reversibility i.e. the automatic generation of specification and design models from Eiffel source, which is of great value when (inevitably) requirements change and the source undergoes

[*]Department of Computer Science, University of Warwick, Coventry CV4 7AL, UK.

maintenance.

However, it is clear that the assertion language of Eiffel is deficient in its support for writing formal specifications. In particular, it has been observed (McKim and Mondou 1993; Waldén and Nerson 1994) that some form of quantification is required. (McKim and Mondou 1993; Waldén and Nerson 1994) have also suggested extensions to the language to introduce quantification. The approach taken here is more analytical, in the sense that we look at the general case, and pay particular attention to the semantics. A comparison with (McKim and Mondou 1993; Waldén and Nerson 1994) is conducted in section 7. The conclusion of this analysis is that a single syntactic form of quantification suffices, where a different semantics may be applied depending on the nature of the type of the variable being quantified. In particular a distinction is drawn between *reference* and *value* quantification, and this matches a distinction between reference and value types, the latter being effected by expanded types in Eiffel. The paper also considers the consequences of this analysis for the run-time checking of quantified assertions. Here the news is good: all forms of reference quantification, and forms of value quantification, where the range of the quantified variable is a clearly identified finite collection of values, may be checked at run-time.

The paper is organised as follows. Section 3 discusses the extension of the assertion language with quantifiers. Section 4 gives a formal semantics to reference types and uses this to discuss the variations in the meaning of reference type quantification. Section 5 does the same for value quantification, which leads to a discussion of the semantics of expanded types. Section 6 discusses the run-time checking of quantified assertions. Section 7 is a summary, and sections 8 and 9 survey related and further work, respectively. An example specification, draughts, is used throughout by way of illustration; its design is outlined in section 2.

2 Example

The ensuing discussion uses a specification of the board game draughts (chequers to some), which has been extracted from a specification compiled under the ISE Eiffel 3 environment. This section describes its basic design.

The design comprises seven classes in addition to the standard library. These are: `COLOUR`, `BOARD`, `PIECE`, `FORWARD_PIECE`, `BACKWARD_PIECE`, `KING_PIECE` and `BOARD_POS`. `COLOUR` defines an enumerated type using the integers, according to the method described by Meyer (Meyer 1988, pp318-20). It is inherited into any class that needs it. `BOARD` is the main class, providing features to query the status of the board and the game, and a command for moving pieces. Some of these features will be introduced and discussed as the paper progresses. `BOARD` is a client of the `..._PIECE`, `PIECE` and `BOARD_POS` classes. Informally specified interfaces of `PIECE` and `BOARD_POS` are given in Figures 1 and 2, respectively.

The `..._PIECE` classes are descendants of `PIECE`, identifying different kinds of piece according to the way in which they can move. The only difference is more restrictive ensure conditions on `normal` and `take_possible`.

`PIECE` reflects the design decision that `BOARD` is responsible for maintaining the rules of draughts as a whole, only delegating to a piece the responsibility of checking whether a move would be valid or not for that piece, under the assumption that other pieces are positioned correctly. An alternative design would be to delegate more responsibility to `PIECE`, in which case `PIECE` would need to know about the positions of other pieces on the board, e.g. through an attribute of type `BOARD` storing the board on which it was placed.

The purpose of `BOARD_POS` is to provide valid board positions on the board, which are pairs of integers on an 8×8 grid. Its specification should be self explanatory. It is an expanded class for reasons that will become clear as the paper develops.

3 Quantified Assertions

An obvious extension to the assertion language (we assume the Eiffel assertion language as defined in (Meyer 1992)) is to add quantifiers, as found in standard predicate logic. Syntactically, adding quantifiers to assertions can be very simple:

```
deferred class interface PIECE

feature -- queries

   colour:INTEGER
         -- attribute stating the colour of the piece
   normal (to:BOARD_POS):BOOLEAN
         -- is the move a normal (not a take) move for the piece?
   pos:BOARD_POS
         -- attribute storing the position of piece on the board
   take_possible (to:BOARD_POS):BOOLEAN
         -- is the move a possible take move for piece, (assuming
         -- other pieces on the board are in appropriate positions)?
   valid_move (to:BOARD_POS):BOOLEAN
      ensure
         result = (normal(to) or take_possible(to))

feature -- commands

   move (bp:BOARD_POS)
      ensure
         pos = bp

end -- class PIECE
```

Figure 1: Outline specification of `PIECE` class

```
   there exists x:C . A
   for all x:C . A
```

where `C` is a class, `A` an assertion (which may also involve quantified expressions), and `x` is a logical variable; `x` must not be used to name any other variable in A or any feature/entity that can be mentioned in `A`, according to the rules for Eiffel assertions (Meyer 1992, pp122-3). A similar syntactic extension is provided in the BON assertion language (Waldén and Nerson 1994).

There are two possible interpretations for such assertions:

1. the variable ranges over possible values that the state of an object of the class may take
2. the variable ranges over instances (direct or indirect) of the class itself

An example of (2) is given by the universal quantification in a possible formalisation of the ensure condition of `win` in `BOARD`:

```
win:BOOLEAN
   ensure
      there exists col:INTEGER .
      for all p:PIECE .
      (p /= Void and then on_board(p))
         implies p.colour = col
```

The ensure condition here may be paraphrased as “all pieces currently on the board are of the same colour”, which corresponds to a winning position in draughts. An example of (1) is given by the universal quantification in a different formalisation of the ensure condition:

```
win:BOOLEAN
   ensure
      there exists col:INTEGER .
      for all pos:BOARD_POS .
      board_status(pos) = Void or else
         board_status(pos).colour = col)
```

which may be paraphrased as “the colour of pieces on occupied board positions is the same”. Here the `for all` quantifier ranges over all pairs of integers that are valid states of board position objects (i.e. within range), and for each one examines `board_status`. Interpretation (1) and (2) will, from now on, be referred to as *value*

```
expanded class interface BOARD_POS

feature -- queries

   col:INTEGER
         -- attribute storing x coord
   mid (to:BOARD_POS):BOARD_POS
      require
         -- 'Current' and 'to' to be on the same diagonal with a position between
      ensure
         -- result is the board_pos between 'Current' and 'to'
   next:BOARD_POS
         -- returns the 'next' board position,
         -- looping round the board from left to right, and bottom to top
   row:INTEGER
         -- attribute storing y coord

feature -- commands

   set (x,y:INTEGER)
      ensure
         col = x and row = y

end -- class BOARD_POS
```

Figure 2: Outline specification of BOARD_POS class

and *reference* quantification, respectively.[1]

A number of questions remain. Are both interpretations required to write useful specifications? If both are required, is there any existing syntax in Eiffel to distinguish the two situations, or is new syntax required? Also, what variations of reference quantification (e.g. only currently existing objects) should be admitted.

Strictly, the answer to the first question could be no, as in practice one might be able to rejig a specification so that all is expressable using quantification by value. However, if one is interested in providing as transparent and natural a specification as possible, this might not be desirable. Looking at the above example, one could attempt to argue about which is more natural than the other (we happen to think it is the first), but that would be missing the point. We believe the specifier should be given the option of writing either, or even both, so that as much information may be provided in the specification in as natural a form as possible.

[1]Note that in both cases the existential quantification is read as *value* quantification.

Furthermore, there do seem to be circumstances in which it is not possible to rewrite the assertion. For example, in the ensure condition for the `move(p:PIECE; to:BOARD_POS)` command in `BOARD`, there must be the conjunct

```
(to.row = 8 or to.row = 1) implies
   ( there exists k:KING_PIECE . new(k)
   and board_status(to) = k
   and k.colour = old colour_to_go
   and k.pos = to and on_board(k)
   and not on_board(p) )
```

which states that if the piece `p` is being moved to the first or last row on the board, then it is replaced with a new[2] king piece whose colour and position is the same as the piece being moved, and `p` is no longer on the board. Here `to:BOARD_POS` is the board position being moved to; `board_status(pos:BOARD_POS):PIECE` is a query which returns the piece on the board position `pos`; and `colour_to_go:INTEGER` is an attribute storing the colour of the next piece to move.

[2]The meaning of `new` is considered in the section on further work.

The only way of expressing this condition without quantification over KING_PIECE (and assuming no other extension of the language), would be to alter the design of the PIECE class to include an attribute identifying the piece as a king, forward moving piece etc. However, this seems to be a rather drastic solution and tends to go against good practice as described in e.g. Meyer's discussion of the use of enumerated types (Meyer 1988, pp318-20)[3].

In order to answer the second question — how, syntactically, should the two types of quantification be distinguished? — one should begin by asking if the existing language already captures the required distinction. In this case, assuming the same syntax for quantification is used, this boils down to distinguishing between the two interpretations by looking at the type chosen for the quantified variable.

Returning to the above example, there is an important difference between e.g. BOARD_POS, for which value quantification seems appropriate, and PIECE, for which reference quantification is desirable. The former is what Cook and Daniels (Cook and Daniels 1994, p75) term *value types* — types which intrinsically are "immutable and lack identity". In particular, entities of value type denote values directly, rather than pointers to values (objects). Thus, it is not surprising that when quantifying over such types one is interested in ranging over values, rather than references to objects — indeed the latter would make no sense!

This suggests that providing a language distinction between reference and value types would be sufficient. In Eiffel, value types (e.g. INTEGER) seem to be characterised using *expanded* types. It is our claim that this captures the required distinction. In section 5, this claim is supported through a detailed semantic argument. First, however, we discuss the various possibilities for assigning a meaning to reference quantification.

[3]An enumerated type would be required to distinguish between different kinds of pieces

4 Reference Quantification and Reference Types

It was mentioned above that there are a number of possible interpretations of reference quantification. These depend on whether:

- objects that exist, will exist or have existed should be considered;
- Void is a valid instantiation of the logical variable.

Before exploring the possibilities further, and in order to add a degree of precision to the discussion, a formal rendering of this semantics, in terms of first order predicate logic (FOPL), is now provided.

4.1 Semantics of reference types

Our approach to semantics is similar to that of Larch (Guttag, Horning, and Wing 1985), where a specification language is given a semantics essentially in terms of theories of order sorted first order predicate logic (OSFOPL). This has the advantage of being widely understood, and it is hoped that such a semantics could provide the basis for proof assistant and simulation tools, such as those described in (Jones, Jones, Lindsay, and Moore 1991; Costa, Cunningham, and Booth 1990). The basic idea is to convert an Eiffel class interface into a theory of OSFOPL. This theory must not only characterise the behaviour of an arbitrary object of the class (i.e. viewing the class as a template) but also the identity and creation of objects. Object identity may be characterised simply in terms of a sort C_{id} corresponding to the collection of all potential instances (i.e. those that have existed, exist, and will exist) of classes conforming to C. Objects have state which may change through time, where the view on this state and ways of changing it are provided through its features. To model this, a sort Σ of possible states is provided. It is assumed that this includes all possible states of a system, only some of which will be abstract states of an object in the system (hence one sort suffices for all class theories and combinations thereof). The values that objects may have in a state are determined by the sort C_s.

To model existence, a boolean function *exists* ranging over object identities, is included. This must also range over Σ, as the existence of an object may change from state to state. So that a different predicate is not required for each class, the sort ANY_{id} is also introduced. This characterises the collection of all potential objects, no matter what the class, and C_{id} is a subsort.

The values that objects may have in a state are modelled by the sort ANY_s. The function

$$state(\Sigma, ANY_{id}) : ANY_s$$

is used to dereference object identities, where $state(s, x)$ returns the value held in the state of x at s. For each class, there is also a sort C_s which is a subsort of ANY_s. In addition, the axiom

$$\forall s : \Sigma; x : C_{id} \cdot state(s, x) : C_s$$

included in the theory for `C`, ensures that appropriate values are assigned to the state of objects of `C`. To summarise, the theory for a class `C` may be presented as

theory C
includes ANY
sorts
  C_{id}, C_s
functions
  ... – feature signatures
axioms
  $C_{id} < Any_{id}$
  $C_s < Any_s$
  $\forall s : \Sigma; x : C_{id} \cdot state(s, x) : C_s$
  $Void : C_{id}$
  ... – axioms for features, invariants etc.

where the line **includes** ANY, means that all the symbols and axioms of ANY are included in C. Here ANY is given by

theory ANY
sorts
  ANY_{id}, ANY_s, Σ
functions
  $exists(\Sigma, ANY_{id}) : BOOLEAN$
  $state(\Sigma, ANY_{id}) : ANY_s$
  $Void : ANY_{id}$
  ... **axioms**
  $\forall s : \Sigma \cdot exists(s, Void)$
  ...

where it is assumed the theory of $BOOLEAN$ is already defined in the usual way (as we assume the logic is already defined). Note that, in the theory ANY, $Void$ is treated as an object (which happens to have no behaviour) and the axiom embodies the assumption that $Void$ always exists. A method by which such theories might be derived from specifications in (extended) Eiffel will be considered in the section on future work.

For simplicity, in giving a semantics to features of a class, we assume a distinction between commands and queries (see e.g. (McKim and Mondou 1993)), and do not allow queries with side-effects (which are visible to the client, so e.g. it is possible to create a new object as part of a query). The distinction is embodied in the specification of draughts used here.

Queries

A query `q(a:A):B` of class `C` introduces the selector function on C_s

$$q(C_s, A) : B$$

in the theory of `C`.

The behaviour of features is obtained by deriving axioms from the require and ensure conditions, and the class invariant. Assuming a class invariant `Inv`, for query `q` as above with require condition `Rq` and ensure condition `Eq`, the derived axiom is[4]

$$\begin{aligned}&\forall s' : \Sigma; v' : C_s; a : A_{id} \cdot \\ &\quad (Rq\langle s', v'\rangle \wedge Inv(s', v')) \Rightarrow \\ &\quad Eq\langle s', v'\rangle[q(v', a) \to Result]\end{aligned}$$

where $[x \to y]$ means that any occurrence of y in the preceding expression is substituted by x. For expression (resp. quantified assertion) `Y`, $Y\langle s', v'\rangle$ is a term (resp. formula) of OSFOPL which is obtained by transforming `Y` as follows:

1. any call to a query of `C`, through expression `p(B)`, is replaced by $p(v', B\langle s', v'\rangle)$

[4]This assumes that `A` is a reference type. s' and v' have been used here to be consistent with the rules when an assertion is a predicate of two states, as discussed in the sequel.

2. any call to a query of another object through `X.p(B)`, where `X` is an expression of reference type, is replaced by $p(state(s', X\langle s', v'\rangle), B\langle s', v'\rangle)$

3. all other expressions translate to themselves without change

(1) ensures that selector functions of the same class are related on the same values (represented here by v'). (2) corresponds to invoking the selector function p on $state(s', X\langle s', v'\rangle)$, the value of the state of the object referenced by X at s', with argument $B\langle s', v'\rangle$. For simplicity, the rules only consider queries with a single argument, though they extend to queries with an arbitrary number of arguments. The rules including expressions of expanded type are given later.

The axiom ensures that the selector function is constrained by its ensure condition only in states where its require condition and invariant hold. To illustrate the process, consider the specification of the query `valid_move` in `BOARD`:

```
valid_move (p:PIECE; to:BOARD_POS):BOOLEAN
   ensure
      result = ( p.colour = colour_to_go
      and (follow_on implies p = last_move)
      and (take_possible implies take(p,to))
      and (not take_possible implies
         p.normal(to)) )
```

The ensure condition states that moving a piece `p` to board position `to` is valid if and only if (conjunct by conjunct): the colour of `p` is the colour of the piece expected to move next; if the move is a follow-on move (e.g. during a multiple take) then the same piece as before is being moved; if a take is possible then the move is indeed a take move; if a take is not possible then the move is a normal move. The fragment of theory that would be generated from this specification is given by figure 3.

Here `BOARD_POS` is translated as the sort $BOARD\_POS_s$ for reasons that will become clear as the paper progresses.

Commands

A command `c(a:A)` introduces the function

$$c(\Sigma, C_{id}, A_{id}) : \Sigma$$

so $c(s, x, a)$ returns the state s' reached by performing c on object x in state s with argument a. Assuming a require condition `Rc` and ensure condition `Ec` on `c`, the derived axiom is

$$\begin{array}{l}\forall s, s' : \Sigma; x : C_{id}; v, v' : C_s; a : A_{id}.\\ \quad (v = state(s, x) \wedge v' = state(s', x)\\ \quad \wedge Inv\langle s, v\rangle \wedge Rc\langle s, v\rangle \wedge c(s, x, a) = s')\\ \qquad \Rightarrow (Inv\langle s', v'\rangle \wedge Ec\langle s, s', v, v'\rangle)\end{array}$$

where for `Y` as before, $Y\langle s, s', v, v'\rangle$ is obtained by transforming `Y` according to the following rules:

1. any call to a query `p(B)` of `C` is replaced by $p(v, B\langle s, s', v, v'\rangle)$

2. any call to a query of another object through `X.p(B)`, is replaced by $p(state(s', X\langle s, s', v, v'\rangle), B\langle s, s', v, v'\rangle)$

3. any call to the old value of a query `old p(B)` of `C` is replaced by $p(v, B\langle s, s, v, v\rangle)$[5]

4. any call to the old value of a query of another object through a call `old X.p(B)`, is replaced by $p(state(s, X\langle s, s', v, v'\rangle), B\langle s, s, v, v\rangle)$

5. all other expressions translate to themselves without change

These conditions extend those given before to deal with the keyword `old` used to refer to the values of queries in the state s from which the command was invoked. An example of the `old` rules in action is given by the translation of another conjunct from `move` in `BOARD`

```
move (p:PIECE; to:BOARD_POS)
   require
      valid_move(p,to)
   ensure
      board_status(old p.pos) = Void
      ...
```

which states that the status of the old position of the piece being moved (i.e. the "from" position) is Void after the move has completed. The definition axiom for *move* derived from this specification fragment follows.

[5]Note that according to (Meyer 1992, p377), old has lower precedence than feature call, hence B is evaluated in the old state i.e. $B\langle s, s, v, v\rangle$.

functions
$\quad valid\_move(BOARD_s, PIECE_{id}, BOARD\_POS_s) : BOOLEAN$
axioms
$$\begin{aligned}
&\forall s' : \Sigma; v' : BOARD_s; p : PIECE_{id}; to : BOARD\_POS_s.\\
&\quad valid\_move(v', p, to) = (colour(state(s', p)) = colour\_to\_go(v')\\
&\qquad \wedge (follow\_on(v') \Rightarrow p = last\_move(v'))\\
&\qquad \wedge (take\_possible(v', p) \Rightarrow take(v', p, to))\\
&\qquad \wedge (\neg take\_possible(v', p) \Rightarrow normal(state(s', p), to)))
\end{aligned}$$

Figure 3: Theory from `valid_move`

$$\begin{aligned}
&\forall s, s' : \Sigma; x : BOARD_{id}; v, v' : BOARD_s;\\
&p : PIECE_{id}; to : BOARD\_POS_s.\\
&\quad (v = state(s, x) \wedge v' = state(s', x)\\
&\quad \wedge valid\_move(v, p, to)\\
&\quad \wedge move(s, x, p, to) = s')\\
&\qquad \Rightarrow (board\_status(pos(state(s, p)) =\\
&\qquad\quad Void \wedge \ldots)
\end{aligned}$$

Clientship and inheritance

Where a class is a client of another class (including library classes), we assume that its theory contains the relevant component, e.g. the interface, of the theory derived from that class, as already outlined. Thus `BOARD` contains the theory of `PIECE`, `KING_PIECE`, `BOARD_POS`, etc.

For inheritance, the theory of the child is an extension of the theory for the parent; that is the child contains the theory of the parent. In addition, there is a subsorting axiom which relates the $-_{id}$ and $-_s$ sorts of the parent and child; this allows for polymorphism by allowing e.g. an entity of sort $KING\_PIECE_{id}$ to be used where an entity of sort $PIECE_{id}$ is expected; similarly for $-_s$ sorts. The details are currently being worked out, including taking account of renaming and repeated inheritance.

4.2 Semantics of reference quantification

Assuming that the quantified assertion appears in the context of $s, s' : \Sigma$ and $v, v' : ANY_{id}$, a naive semantics for reference quantification expressed in this framework would be

$$\exists x : C_{id}.A\langle s, s', v, v'\rangle$$
$$\forall x : C_{id}.A\langle s, s', v, v'\rangle$$

for universal and existential quantification, respectively. However, this ignores any consideration of whether the variable ranges only over objects currently existing, or all potential objects (i.e. including those that have existed or will exist), and whether `Void` is a valid instantiation.

Looking again at the ensure condition, involving reference quantification, for `win` in `BOARD`

```
there exists col:INTEGER .
for all p:PIECE .
   ((p /= Void and then on_board(p))
   implies p.colour = col)
```

it is interesting to note that only pieces in existence need to be considered by the quantification (and this is the interpretation we are assuming here), as the behaviour of `on_board` is only determined for arguments referencing existing objects. Since, in general, entities can only refer to existing objects (or be Void), this should be the semantics of reference quantification. By a similar argument, `Void` should be an allowable instantiation, as features may have Void arguments and results[6]. Thus the semantics for reference quantification should be: consider only currently existing objects, and allow `Void` as an instantiation.

Assuming that the quantified assertion appears in the context of $s, s' : \Sigma$ and $v, v' : ANY_{id}$,[7] the preferred semantics may be expressed as

$$\exists x : C_{id} \cdot (exists(s', x) \wedge (A\langle s, s', v, v'\rangle))$$

[6]Though note that we have had to disallow `Void` explicitly here, because it would be disallowed by the require condition of `on_board`.

[7]This context, of course, would only appear in an ensure condition of a command. The semantics in a context of only $s' : \Sigma$ and $v' : ANY_{id}$ follows similarly.

$$\forall x : C_{id} \cdot (exists(s', x) \Rightarrow (A\langle s, s', v, v'\rangle))$$

Clearly alternative semantics could be provided by leaving out the $exists(s', x)$ constraint (all potential objects are to be considered), or inserting the additonal condition that $x \neq Void$. By this semantics, the axiom derived from the specification of `win` using the above ensure condition is

$$\begin{array}{l} \forall s' : \Sigma; v' : BOARD_s \cdot \exists col : INTEGER \cdot \\ \forall p : PIECE_{id} \cdot \\ \quad exists(s', p) \Rightarrow \\ \qquad ((p \neq Void \wedge on\_board(v', p)) \Rightarrow \\ \qquad\quad colour(state(s', p)) = col) \end{array}$$

assuming, for the time being, that `INTEGER` is treated like `BOOLEAN`, in that we assume its theory is pre-defined in the logic.

A similar semantics has been chosen for quantifiers in the POOL assertion language (America and de Boer 1990), though we note that they only have reference quantification.

5 Value Quantification and Expanded Types

Recall that for value quantification the interpretation of e.g.

```
   for all x:C . A
```

is to quantify over all possible values that the state of an object of `C` might have. The semantics of this is simply

$$\forall x : C_s \cdot A\langle s, s', v, v'\rangle$$

assuming a context of $s, s' : \Sigma$ and $v, v' : ANY_s$, noting that the rules for $Y\langle s, s', v, v'\rangle$ need to be extended to include expanded types. To illustrate this consider again the motivating example

```
win:BOOLEAN
   ensure
      there exists col:INTEGER .
      for all pos:BOARD_POS .
      board_status(pos) = Void or else
      board_status(pos).colour = col)
```

from which the axiom

$$\begin{array}{l} \exists col : INTEGER \cdot \forall pos : BOARD\_POS_s \cdot \\ \quad board\_status(s', v', pos) = Void \vee \\ \quad colour(s', state(s', board\_status(s', v', pos))) \\ \qquad = col \end{array}$$

is derived (assuming that there is no class invariant), where, for this example, the rules already defined for $Y\langle s, s', v, v'\rangle$ are sufficient. However, in general the rules need to be extended to deal with expanded types. Consider for example the kinging clause, introduced earlier, in the ensure condition of `move` in `BOARD`.

```
   (to.row = 8 or to.row = 1) implies ...
```

Assuming the same rules for expanded as those for reference types, `to.row` would be translated as $row(state(s', to))$ which is incorrect because to would be of sort $BOARD\_POS_s$ not $BOARD\_POS_{id}$, as required by $state$. This is actually not a problem with quantification, but with the interpretation of calls, when the entity being called is of type ANY_s. It is simple to rectify: just reinterpret the invocation in such situations, so that it does not dereference entities of sort ANY_s. Then, for this example, the translation is simply $row(to)$, as required. The general rules are given in the sequel.

Returning again to the question of providing a syntactic distinction between reference and value quantification, it should already be clear that `expanded` types in Eiffel provide the solution. That is, when the type of the quantified variable is expanded the interpretation of quantification is value quantification, otherwise it is reference quantification. An expanded type is a type whose syntactic representation is `expanded C` where `C` is a class (or `C` where `C` is an expanded class). It is a promising candidate because an entity of expanded type directly denotes a value, rather than a reference to some object whose state has that value. This allows equality and assignment to directly work with values, giving a more natural treatment of (value) types such as the integers. In the next section, a semantics of expanded types is developed, which both supports the interpretation of value quantification given above and matches their use in Eiffel.

5.1 Semantics of expanded types

From the discussion so far, the treatment of expanded types is different from the treatment of reference types in at least the following two ways:

- Wherever an entity is declared to be of expanded type in a specification, it is declared to be of $-_s$ sort in the corresponding logical theory.
- A call of the form `X.p(b)`, where `X` is of expanded type is interpreted so that X is not dereferenced.

The latter is captured formally in the complete interpretation of $A\langle s,s',v,v'\rangle$, for some assertion expression `A`, as follows:

1. any call to a query `p(B)` of `C` is replaced by $p(v, B\langle s,s',v,v'\rangle)$
2. any call to a query of another object through a `X.p(B)`, is replaced by
   (a) $p(state(s', X\langle s,s',v,v'\rangle), B\langle s,s',v,v'\rangle)$ if `X` is of reference type
   (b) $p(X\langle s,s',v,v'\rangle, B\langle s,s',v,v'\rangle)$, if `X` is of expanded type
3. any call to the old value of a query `old p(B)` of `C` is replaced by $p(v, B\langle s,s,v,v\rangle)$
4. any call to the old value of a query of another object through `old X.p(B)`, is replaced by
   (a) $p(state(s, X\langle s,s',v,v'\rangle), B\langle s,s,v,v\rangle)$, if `X` is of reference type
   (b) $p(X\langle s,s',v,v'\rangle, B\langle s,s,v,v\rangle)$, if `X` is of expanded type

Changes have been made to conditions (2) and (4). In each case, a different interpretation is chosen depending on whether `X` is of reference or expanded type. If the former, then the condition is as before; if the latter, then $X(s,s',v,v')$ is no longer dereferenced using the *state* function.

This semantics is in agreement with Eiffel expanded types for equality and reattachment, because entities of expanded type directly denote values: thus `x = y` means that the value of `x` is equal to the value of `y`, not that they point to the same object; and `x := y` means that the value of `x` in the state reached after the assignment is the value of `y` before the assignment. However, the semantics of commands needs to be considered more carefully. According to the semantics for reference types, the command `set` in `BOARD_POS` would be represented by the function

$$\begin{aligned}&set(\Sigma, BOARD\_POS_{id},\\&\quad INTEGER, INTEGER) : \Sigma\end{aligned}$$

with defining axiom

$$\begin{aligned}&\forall s,s' : \Sigma; b : BOARD\_POS_{id};\\&\quad v,v' : BOARD\_POS_s; x,y : INTEGER\cdot\\&\quad (v = state(s,b) \land v' = state(s',b) \land\\&\quad x >= 1 \land x <= 8 \land y >= 1 \land y <= 8 \land\\&\quad set(s,b,x,y) = s') \Rightarrow\\&\qquad (col(v') = x \land row(v') = y)\end{aligned}$$

Now consider, for example, the call `b.set(1,1)` where, for simplicity, `b:BOARD_POS` is assumed to be an attribute of the calling object. According to the semantics for expanded types proposed above, $b(\ldots)$ would be of sort $BOARD\_POS_s$. Thus the translation of this call, assuming that s is the state in which the call is made and w is the value in s of the object making the call, into the expression $set(s, b(w), 1, 1)$, would be invalid as *set* expects its second argument to be of sort $BOARD\_POS_{id}$.

So let's suppose we change the semantics of commands for expanded types so that `set` is now represented by the function

$$\begin{aligned}&set(\Sigma, BOARD\_POS_s,\\&\quad INTEGER, INTEGER) : \Sigma\end{aligned}$$

This solves the above problem, but now the defining axiom needs to be changed. Changing it in the most obvious way would result in

$$\begin{aligned}&\forall s,s' : \Sigma; v,v' : BOARD\_POS_s;\\&x,y : INTEGER\cdot\\&\quad (x >= 1 \land x <= 8 \land y >= 1 \land y <= 8 \land\\&\quad set(s,v,x,y) = s')\\&\qquad \Rightarrow (col(v') = x \land row(v') = y)\end{aligned}$$

The main difference here is that the values v and v' are no longer obtained by dereferencing some b. The problem with this axiom is that, although the value v is passed as an argument to set, the value v' is not. Thus, looking again at the call `b.set(1,1)` and its translation as $set(s, b(w), 1, 1)$, where s is the state from which the call is made and w the value of the calling object in s, we see that the above axiom does not ensure that, for example, $col(b(w')) = 1$, where w' is the value of the state of the calling object in the state reached. To put it another way, `b` holds a state value rather than a pointer to a state value. The effect of `b.set(1,1)` is to change the value held by `b` (i.e. the state of the calling object) rather the value of the state of the object pointed to by `b`. The point of the argument above is that, assuming the above function chosen to represent `set` in the semantics, there is no way of defining the required behaviour in the axiom derived from its specification. This is because there is no way of accessing the value held by `b` in the state reached by performing the command.

The way out of this is to change the function used to represent a command, which, in this case may be achieved simply by returning a value of sort $BOARD\_POS_s$ as part of the result. Thus the function is

$$\begin{array}{l} set(\Sigma, BOARD\_POS_s, \\ \quad INTEGER, INTEGER) : \\ \qquad \Sigma \times BOARD\_POS_s \end{array}$$

with defining axiom

$$\begin{array}{l} \forall s, s' : \Sigma; v, v' : BOARD\_POS_s; \\ x, y : INTEGER. \\ \quad (x >= 1 \wedge x <= 8 \wedge y >= 1 \wedge y <= 8 \wedge \\ \quad set(s, v, x, y) = (s', v')) \\ \qquad \Rightarrow (col(v') = x \wedge row(v') = y) \end{array}$$

Then the call `b.set(1,1)` becomes $b(w') = snd(set(s, b(w), 1, 1))$ where s is the state in which the call is made, w is the value of the state of the calling object in s, and w' is the value of the state of the calling object in $fst(set(s, b(w), 1, 1))$. fst and snd return the first and second items of a pair, respectively.

This semantics also works for reference types, by changing the semantics of calling to dereference before making the call, rather than dereference as part of the call. For example, if `BOARD_POS` was not expanded, then the above call would translate to

$$\begin{array}{l} state(s', b(w')) = \\ \quad snd(set(s, state(s, b(w)), 1, 1)) \end{array}$$

with s, w and w' as above, and where $s' = fst(set(s, state(s, b(w))))$. That is, $b(w)$ is dereferenced before being passed as an argument to set.

Generalising this, the semantics of a command `c(a:A)` with require condition `Rc` and ensure condition `Ec` in a class with invariant `Inv` is given by the theory fragment in figure 4.

6 Run-time Assertion Checking

A key motivation for introducing assertions into Eiffel is that they can be monitored at run-time. This means that as a program is being executed, require conditions and class invariants are evaluated when a feature is invoked (by a client, not `Current`) and ensure conditions and invariants are evaluated on completion of a routine's execution. An exception is raised if an assertion is violated. This section considers the run-time evaluation of quantified assertions.

6.1 Reference quantification

In short, it should not be difficult to check *all* reference quantified assertions at run-time. The semantics given above only allows quantification of currently existing objects (including the value `Void`). Since this will always be a finite collection, and, presumably, the run-time system will have access to all members of this collection, an assertion checking mechanism can, in the worst case, evaluate the assertion for all existing objects in turn. Of course the mechanims may be made more efficient by only considering those objects which conform with the type of the quantified variable. No doubt further optimisations could be made by examination of the assertion. For example, in checking the assertion

```
for all x:PIECE . c.has(x) implies ...
```

functions
$c(\Sigma, C_s, A_{id}) : \Sigma \times C_s$
axioms
$\forall s, s' : \Sigma; v, v' : C_s; a : A_{id}.$
$(Inv\langle s, v\rangle \wedge Rc\langle s, v\rangle \wedge c(s, v, a) = (s', v')) \Rightarrow (Inv\langle s', v'\rangle \wedge Ec\langle s, s', v, v'\rangle)$

Figure 4: Theory fragment for a command

where `c:COLLECTION[PIECE]`, it would only be necessary for the checking mechanism to consider those objects actually stored in the collection. This is also noted in (Waldén and Nerson 1994, pp53-7), where they give a special syntax for this case.

6.2 Value quantification

The situation here is more complicated. In general, to check a value quantified expression requires a search through all possible values that the state of an object of the class could have. Since, with assertion checking one is dealing with entities which are storable in a computer's memory, one may assume that the number of such values, though large, will be finite. Thus in theory this should be possible. The problem is in knowing what are the valid values of the expanded type. One solution to this would be to keep copies of all possible values in some allocated area of storage. Not only is this space-inefficient, but it also reduces the problem to one of providing a way of generating the values in the first place. The other option is to generate the values dynamically. Thus either way, a method for generating the values is required. One possibility is to 'hardwire' such generation procedures; whilst this may be an adequate solution for basic expanded types such as the integers, it is clearly not the ideal solution for expanded types such as `BOARD_POS`. The remainder of this section outlines an alternative approach.

Consider again the quantified assertion in `win`. There we are wishing to quantify over all values. One way to achieve this would be to start with the value (`row=1`,`col=1`) and continue to invoke the `next` feature to obtain successive values, until (`row=8`,`col=8`) was reached—i.e. whilst the value being obtained was between these two values. If the first of these values was referred to as `first_pos` and the second as `last_pos`, then we might write the collection defined by this process as `(first_pos,last_pos,next,<)` where `infix "<" (y:BOARD_POS):BOOLEAN` would be a new feature of `BOARD_POS` saying what it means for one `BOARD_POS` value to be before another. This in turn suggests an extended syntax for value quantification, for example

```
for all b:BOARD_POS in
   (first_pos,last_pos,next,<) .  ...
```

requiring `b` to be chosen from the designated collection. It remains to say how this translates into our semantics and how `first_pos` and `last_pos` may be defined in the class. The former may be achieved by providing a constructor function for returning a value of type `BOARD_POS`: that is, we would like to have functions `first_pos,last_pos` which can be used in expressions to represent these values (much like `1`, `2`, etc. represent integer values). In current Eiffel, these may be defined in the class in which they are required. If that class is the expanded class itself, then this would mean that all values of the type would themselves have the constructors as features.

A better solution, in our view, would be to provide a way of defining constructors in the expanded class, without the latter side effect. This could be achieved through new syntax. Perhaps a more efficient solution would be to allow the interpretation of certain commands as constructors. For example, the `set` command in `BOARD_POS` only updates attributes of the class, and, in addition, needs to make no reference to previous values of these attributes. The former means that it does not need to return a new state [8] and the latter that it does not need to be passed the old value of the state as argument.

[8] If it referred to the state of supplier objects, then it could have side effects in those objects, which could, in theory be experienced by the calling object.

Thus it could be interpreted by the theory fragment in figure 5

which would allow `set` to be used in expressions. Under this scheme the quantified assertion above could now be expressed as

```
   for all b:BOARD_POS in
      (set(1,1),set(8,8),next,<) .  ...
```

With regard to semantics, the above quantified expression could be interpreted as

$$\forall b : BOARD\_POS_s \cdot \\ \quad in\_this\_collection(b) \Rightarrow \ldots$$

where *in_this_collection* is a recursive function defined in terms of *set*, *next* and $<$, in the usual way.

7 Summary

Two types of quantification have been introduced, namely reference and value quantification. These are distinguished syntactically, by examining whether the quantified variable is of reference or expanded type. A semantics for reference and expanded types has been given, and this has then been used to provide a semantics for the two forms of quantification. The run-time checking of quantified assertions has also been considered. Here the news is good: all forms of reference quantification, and forms of value quantification, where the range of the quantified variable is a clearly identified finite collection of values, can be checked. A systematic approach to defining this collection has been proposed.

8 Related Work

The work described here is related, in general, to work on OO specification languages such as those described in (Lano and Haughton 1994). We restrict ourselves here to consider specific proposals to extend Eiffel with quantifiers.

The idea of extending Eiffel with quantification is not new. It is mentioned in (Meyer 1994), and proposals appear in (McKim and Mondou 1993) and (Waldén and Nerson 1994). Meyer discounts quantifiers on the grounds that they are not expressive enough e.g. to specify acyclicity of linked lists (this is not a first order property). Instead, Meyer proposes the use of an intermediate functional language. Whilst this may be a valid point, it misses the fact that quantifiers often provide a way of expressing properties naturally. Thus, whilst it may be possible to express the required property in terms of a recursive function, this may not always be a desirable thing to do.

McKim proposes two kinds of quantifier, one which ranges over integers between a designated range, and another which ranges over objects of any type. The distinction is made, as, he argues, the former is simple enough to be compiled and checked at run-time, whereas the latter is too general to be useful for run-time checks. In our terms, McKim's quantification over an integer range is a particular form of value quantification. His second form of quantification seems to encompass reference quantification and all other forms of value quantification.

The BON notation (the analysis and design notation and process for the Eiffel method) also extends the Eiffel assertion language with syntax for universal and existential quantifiers. Their purpose is not so much to provide the ability to specify complete contracts (McKim's motivation) but rather to extend Eiffel into a wide-spectrum notation, suitable for high-level system specification. Again, no formal semantics is provided, and no distinction between reference and value quantification is made. Run-time checking of quantified assertions is only considered for reference quantification.

Thus our proposal is more general than previous proposals and recognises a distinction (between value and reference quantification) which has been previously ignored. This insight has led to a clarification of the semantics of quantified assertions, which we have been able to express in detail and precisely. It has also clarified which forms of quantified assertions can be checked at run-time.

9 Further work

This paper has discussed the extension of Eiffel with quantification, to enhance its specification

functions

$set(\Sigma, INTEGER, INTEGER) : BOARD\_POS_s$

axioms

$\forall s : \Sigma; v : BOARD\_POS_s; x, y : INTEGER.$

$(Rset(s, v) \wedge set(s, x, y) = v) \Rightarrow Eset(s, v))$

Figure 5: Theory for `set` command as a constructor

capability. The further work described here is restricted to consideration of further enhancements.

Expanded classes model *value types*, but do not allow for their full specification; in particular it is difficult to define constructors. We have suggested how some commands of expanded classes could be reinterpreted as constructors and used as such in expressions. This would effectively give expanded classes the full power of ADT specification, as it would provide constructors for use in invariants. This would be useful in formulating designs (*viz.* `BOARD_POS` in draughts), and could provide a means by which full contract specifications can be provided for a reference type, by building it in terms of some appropriate value type. However, this proposition needs to be checked, including some consideration of the constraints placed on implementations of classes specified using such an approach. If full contract specifications of value types could be provided, then there would be no need to define basic types such as $BOOLEAN$ and $INTEGER$ in the logic, as their behaviour could be derived from their Eiffel specifications using our semantics. This would improve seamlessness, as reasoning would then be based entirely on what is specified in the language.

Although Eiffel provides all the necessary *creation* procedures for writing programs, it provides no way of talking about creation in the assertion language. In particular, it has no way of ensuring that an object is one that has just been created, so can not be the same as any other currently existing object. This seems to be a deficiency; for example, it was observed, in the discussion of quantification in Section 3, that the kinging condition in the ensure condition for `move` from `BOARD` requires one to be able to say that an object is 'new', in the sense that it did not exist when the command was called, but does now exist. A small extension to the language suffices: a new keyword `new`, which has a single argument of type `ANY`, where `new(x)` has the semantics $(\neg exists(s, x) \wedge exists(s', x))$, s and s' being the state before and after the command, respectively.

A specification language needs to provide support for expressing *frame conditions*—stating what does not change when a command is performed. For example, the popular specification languages VDM-SL and Z provide support e.g. through the `ext rd`, `ext wr` and Δ, Ξ notation, respectively. Eiffel provides comparable support through the use of `strip` expressions in `ensure` conditions, which indicate which parts of the state of the current object do not change. However, for an object-oriented specification language, these are not sufficient, since, to be complete, one must also be able to express how a command affects the state of supplier objects. `strip` can only be used to state that attributes of the current object are not reattached, not that objects they are attached to have not changed state. Thus some modifications or extensions must be made to the assertion language to enable the specification of frame conditions.

One solution would be to extend the scope of `strip`, to consider supplier objects. However, this would lead to a proliferation of `strip` statements, one for each supplier. However, this could be combined with a notation which lists those objects whose state is changed by a command. For example, the Larch approach to specifying object systems (Liskov and Wing 1993) allows a clause consisting of the keyword `modifies` followed by a list of object reference valued expressions (e.g. `modifies e1,e2`) to be included in the definition of a feature. The semantics of this clause is that only the states of the objects denoted by the expressions `e1` and `e2` can be changed; all other objects have unchanged states. The Eiffel assertion language could be extended in a similar way.

References

America, P. and F. de Boer (1990). A Sound and Complete Proof System for SPOOL. Technical Report 505, Philips Research Labs.

Booch, G. (1994). *Object-Oriented Analysis and Design, with Applications.* Benjamin Cummings.

Cook, S. and J. Daniels (1994). *Designing Object Systems.* The Object-Oriented Series. Prentice Hall.

Costa, M., J. Cunningham, and J. Booth (1990). Logical Animation. In *Proceedings of the International Conference on Software Engineering.*

Guttag, J., J. Horning, and J. Wing (1985). The Larch Family of Specification Languages. *IEEE Software 2*(5), 24–36.

Jones, C. B. (1990). *Systematic Software Development using VDM* (second ed.). Prentice Hall.

Jones, C. B., K. Jones, P. Lindsay, and R. Moore (1991). *Mural: A Formal Development Support System.* Springer Verlag.

Lano, K. and H. Haughton (Eds.) (1994). *Object-Oriented Specification Case Studies.* The Object-Oriented Series. Prentice Hall.

Liskov, B. and J. Wing (1993). Specifications and their use in Defining Subtypes. In *Proceedings of OOPSLA-93.*

McKim, J. and D. Mondou (1993). Class Interface Design: Designing for Correctness. *Journal of Systems and Software 23*(2), 85–92.

Meyer, B. (1988). *Object-Oriented Software Construction.* Prentice Hall.

Meyer, B. (1992). *Eiffel: The Language.* The Object-Oriented Series. Prentice Hall.

Meyer, B. (1994). Beyond Design by Contract. Keynote Lecture at TOOLS Pacific 94.

Waldén, K. and J. Nerson (1994). *Seamless Object-Oriented Software Architecture: Analysis and Design of Reliable Systems.* The Object-Oriented Series. Prentice Hall.