Model-based specification

Lex Bijlsma
Universiteit Utrecht, Department of Information and Computing Sciences
P.O. Box 80089, 3508 TB Utrecht, The Netherlands
e-mail lex@cs.uu.nl

May 1, 2000

1 Introduction

Procedures in an imperative programming language are specified by means of a precondition and a postcondition, expressed in the procedure's parameters and any global variables that are referenced. A simple way of looking at methods in object-oriented languages, suggested by early implementations, is to regard these as procedures with one special parameter that is written before the method name in method calls. This parameter is not explicitly present in the method's definition, but can be retrieved via the keyword **this** or **self**.

The specification style suggested by this view is the use of pre- and postconditions expressed in both **this** and the method's explicit parameters. Here the question arises what kind of assertions are permitted about the state of **this** and other objects among the parameters. The simplest way to view object types, as made explicit in Oberon-2 [12], is as a special kind of record types; consistent with such a view is the characterization of object states by means of field (or attribute) values. Indeed, many specification methodologies for object-oriented programs in the literature (for instance [1]) proceed this way.

Yet, one might argue that such an approach does not go well with the concepts of information hiding and subtype polymorphism inherent in object-oriented design. It conflicts with information hiding because that principle states that the object's clients should not even be aware of the existence of private fields, let alone be dependent on them for method specifications. It conflicts with subtype polymorphism because the procedure body could at run-time be replaced with a fresh implementation that produces similar observable effects by a completely different set of private fields. Finally, specifications based on field values become patently impossible as soon as one introduces abstract classes or interfaces, which may not have any fields at all.

The way out of this dilemma is to refrain completely from using fields in specifications. In their stead one might use an axiomatic characterization of the required behavior [9, 5], but in practice it is often very difficult to find suitable axiomatizations. An approach based on data refinement (e.g. [8]) is easier to apply. In this note, it is shown how one methodology of this type may be integrated with Dijkstra-style program derivation. Special attention is paid to the technical problems presented by dealing with specification variables representing initial states at different levels of abstraction.

2 Models

An interface or pure abstract class is the programming-language counterpart of an abstract datatype. Following the principle of *direct mapping* [9], we assume that the abstract datatype's purpose is to describe some entity from the problem domain whose behavior is known. Therefore we propose to express its specification in terms taken from the problem domain.

As an example, consider an abstract datatype *DateSet* whose values describe a set of integers from the range $1 \mathinner{..} 31$. Here the problem domain whose language we use for specification purposes is mathematics, but in different examples we might use musical notation, grammars, pictures or chemical formulae. The mathematical set so described is called the *model* of the abstract datatype. In this case, the model is described by a singe *model variable*, say a, of type $\mathcal{P}(\mathbb{Z})$; in more realistic examples, one has a great number of model variables – that is to say, the model traverses a cartesian product of high dimension. The restriction that the members of the set are taken from the range $1 \mathinner{..} 31$ could, in this case, be coded into the type of a. For dependencies between several model variables, this becomes artificial; therefore, it is preferable to express such constraints in a separate *model invariant*, in the present case

$$M: \quad \langle \forall i : i \in a : 1 \leq i \leq 31 \rangle \,.$$

The model invariant is silently added to the pre- and postconditions of all methods.

3 Initial values

Consider a statespace Ω. Traditionally, we characterize statements over Ω by a pair of predicates over Ω: the Hoare triple notation

$$\{U\} \;\; S \;\; \{V\} \tag{1}$$

signifies

$$[U \;\Rightarrow\; wp.S.V] \,,$$

where the brackets denote universal quantification over Ω. For many purposes, however, it is more practical to use a convention where the postcondition V is not a predicate over Ω but rather over $\Omega \times \Omega_\bullet$, where $\Omega_\bullet$ is a disjoint copy of Ω. Let us call this an *extended* predicate, as opposed to a *plain* predicate that is defined over Ω. If x is the coordinate vector of Ω and $x_\bullet$ that of $\Omega_\bullet$, extended predicates contain both x and $x_\bullet$. The idea is that $x_\bullet$ is used to record the initial value of x. For instance, a statement that squares the value of x may be specified as

$$\{\mathit{true}\} \;\; S \;\; \{x = x_\bullet^2\} \,.$$

A similar notation is used, for instance, in the postcondition notation of the Eiffel programming language [9] (where $x_\bullet$ is called **old** x) and in the OCL specification language [13] (where $x_\bullet$ is called x**@pre**). Its usefulness lies in the fact that it shortens specifications by doing away with the need for most specification variables, without having to drag in the full apparatus of the relational calculus. The extent to which this simple expedient cleans up formulae becomes clear if we observe that the classical criterion for repetition termination [3, (9, 28)]

$$\langle \forall \tau :: \{P \wedge B \wedge t = \tau\} \;\; S \;\; \{P \;\wedge\; t < \tau\} \rangle$$

can now be shortened to

$$\{P \wedge B\} \;\; S \;\; \{P \;\wedge\; t < t_\bullet\} \,. \tag{2}$$

As suggested by these examples, the intention may be captured by considering $x_\bullet$ as a specification variable and implicitly extending the precondition with a conjunct $x = x_\bullet$. In other words, (1) may be defined for extended V by

$$\langle\, \forall x_\bullet :: [U \;\wedge\; x = x_\bullet \;\Rightarrow\; wp.S.V] \,\rangle \,, \tag{3}$$

where the brackets still quantify over Ω. Some applications of this approach were demonstrated in [4].

4 Weakening postconditions

The classical theorem about weakening postconditions reads

$$\{U\} \;\; S \;\; \{V\} \;\wedge\; [V \Rightarrow W] \;\Rightarrow\; \{U\} \;\; S \;\; \{W\} \,. \tag{4}$$

Obviously this must be adjusted in the case where V and W are extended predicates, for instance because one must decide what to do with the free $x_\bullet$ now occurring in $[V \Rightarrow W]$. The most obvious solution would be to replace (4) with

$$\{U\} \;\; S \;\; \{V\} \;\wedge\; [V \Rightarrow W]' \;\Rightarrow\; \{U\} \;\; S \;\; \{W\} \,, \tag{5}$$

where the Hoare triples are interpreted as discussed above, and again the primed brackets quantify over $\Omega \times \Omega_\bullet$. However, it turns out that (5), although valid, is too weak to be useful. We shall now derive a better replacement. In the derivation below, $U_\bullet$ is an abbreviation for $(x := x_\bullet).U$.

$$
\begin{array}{ll}
 & \{U\} \;\; S \;\; \{W\} \\
\equiv & \quad \{(3) \text{ with } V := W\} \\
 & \langle\, \forall x_\bullet :: [U \;\wedge\; x = x_\bullet \;\Rightarrow\; wp.S.W] \,\rangle \\
\equiv & \quad \{\, [U \wedge x = x_\bullet \;\Rightarrow\; U_\bullet] \text{ by instantiation and one-point rule}\} \\
 & \langle\, \forall x_\bullet :: [U \;\wedge\; U_\bullet \;\wedge\; x = x_\bullet \;\Rightarrow\; wp.S.W] \,\rangle \\
\equiv & \quad \{\text{shunting}\} \\
 & \langle\, \forall x_\bullet :: [U_\bullet \;\Rightarrow\; (U \;\wedge\; x = x_\bullet \;\Rightarrow\; wp.S.W)] \,\rangle \\
\equiv & \quad \{\, U_\bullet \text{ is independent of state}\} \\
 & \langle\, \forall x_\bullet :: U_\bullet \;\Rightarrow\; [U \;\wedge\; x = x_\bullet \;\Rightarrow\; wp.S.W] \,\rangle \\
\equiv & \quad \{\text{trading}\} \\
 & \langle\, \forall x_\bullet : U_\bullet : [U \;\wedge\; x = x_\bullet \;\Rightarrow\; wp.S.W] \,\rangle \\
\Leftarrow & \quad \{\text{transitivity}\} \\
 & \langle\, \forall x_\bullet : U_\bullet : [U \;\wedge\; x = x_\bullet \;\Rightarrow\; wp.S.V] \;\wedge\; [wp.S.V \;\Rightarrow\; wp.S.W] \,\rangle \\
\Leftarrow & \quad \{\text{monotonicity of } wp\,\} \\
 & \langle\, \forall x_\bullet : U_\bullet : [U \;\wedge\; x = x_\bullet \;\Rightarrow\; wp.S.V] \;\wedge\; [V \Rightarrow W] \,\rangle \\
\equiv & \quad \{\text{term split}\} \\
 & \langle\, \forall x_\bullet : U_\bullet : [U \;\wedge\; x = x_\bullet \;\Rightarrow\; wp.S.V] \,\rangle \;\wedge\; \langle\, \forall x_\bullet : U_\bullet : [V \Rightarrow W] \,\rangle \\
\Leftarrow & \quad \{\text{formal weakening of domain}\} \\
 & \langle\, \forall x_\bullet :: [U \;\wedge\; x = x_\bullet \;\Rightarrow\; wp.S.V] \,\rangle \;\wedge\; \langle\, \forall x_\bullet : U_\bullet : [V \Rightarrow W] \,\rangle \\
\equiv & \quad \{(3)\} \\
 & \{U\} \;\; S \;\; \{V\} \;\wedge\; \langle\, \forall x_\bullet : U_\bullet : [V \Rightarrow W] \,\rangle \,.
\end{array}
$$

This shows that a better replacement for (5) is

$$\{U\}\ \ S\ \ \{V\}\ \ \wedge\ \ \langle\, \forall x_\bullet : U_\bullet : [V \Rightarrow W]\,\rangle\ \ \Rightarrow\ \ \{U\}\ \ S\ \ \{W\} \tag{6}$$

As an example, we observe that

$$\{x > 0\}\ \ S\ \ \{x = x_\bullet\}$$

implies

$$\{x > 0\}\ \ S\ \ \{x > 0\}\,,$$

because

$$\langle\, \forall x_\bullet : x_\bullet > 0 : [x = x_\bullet\ \Rightarrow\ x > 0]\,\rangle\,.$$

In this case the stronger requirement from (5), viz.

$$[x = x_\bullet\ \Rightarrow\ x > 0]'$$

does not hold.

(At this point, it may be remarked in passing that the theorem about strengthening preconditions carries no surprises of this kind and is just what you would expect it to be.)

As a practical matter, if the number of model variables is large, most of these will not be changed by a given method call. Expressing this explicitly leads to a large number of conjuncts of the form $b = b_\bullet$ the postconditions. This can be avoided by adding to each specification a *modification clause* stating explicitly which model variables may be changed. For every b not in the modification clause, a conjunct $b = b_\bullet$ is added silently to the postcondition. This technique is known as *framing* [6].

For an example, consider a method *add* for the abstract data type *DateSet* that, given an integer parameter n, will add n to a. Its specification has modification clause a, precondition $1 \le n \le 31$, and postcondition $a = a_\bullet \cup \{n\}$.

5 Implementation

An implementation of an interface introduces private fields in order to simulate the operations on the model variables. For instance, the set a from the *DateSet* model may be simulated by a boolean array f. The relationship between these implementation fields and the model variables is expressed in an *implementation invariant*, in this case

inv I: $\langle\, \forall i : 1 \le i \le 31 : f[i-1]\ \equiv\ i \in a\,\rangle$.

Many authors, e.g. [8], require that the implementation invariant constitute a partial mapping ('abstraction function') from the implementation fields to the model variables. In other words, they demand that any state of the implementing object will correspond to a single state of the model. This restriction, however, is too strict in case some aspects of the model cannot be retrieved by the methods specified. In [4], a realistic example of this phenomenon is presented: the dichotomy between input and output files in Pascal that does not have to be implemented if the underlying file system permits mixing read and write operations on the same file.

The implementation will provide a body for every method specification in the interface. The relationship between a specification with precondition U and postcondition V, both expressed in the model variable a, on the one hand, and the body S, expressed in the field f, on the other, should be described by a *coordinate transformation* [2], also known as *data refinement* [10, 11]. That is to say, for model invariant M and implementation invariant I, the implementation body S should satisfy

$$\langle\, \forall a_\bullet :: \{\langle\, \exists a :: M \wedge I \wedge U \wedge a = a_\bullet \,\rangle\}\ \ S\ \ \{\langle\, \exists a :: M \wedge I \wedge V \,\rangle\}\,\rangle\ .$$

Using the one-point rule, we may simplify this to

$$\langle\, \forall a_\bullet :: \{(a := a_\bullet).(M \wedge I \wedge U)\}\ \ S\ \ \{\langle\, \exists a :: M \wedge I \wedge V \,\rangle\}\,\rangle\ .$$

In order to avoid carrying along the existential quantification, it is advisable to start by constructing a program fragment T over the state space spanned by a and f together that satisfies

$$\{M \wedge I \wedge U\}\ \ T\ \ \{M \wedge I \wedge V\}$$

and, furthermore, has the property that a does not occur in any statement that may change the value of f. Then S can be obtained from T by simply leaving out all statements that refer to a.

In our running example, the body of method *add* may be determined by constructing an T satisfying

$$\{M \wedge I \wedge 1 \le n \le 31\}\ \ T\ \ \{M \wedge I \wedge a = a_\bullet \cup \{n\}\}\ ,$$

with

$$\begin{aligned} M &: \langle\, \forall i : i \in a : 1 \le i \le 31 \,\rangle\ , \\ I &: \langle\, \forall i : 1 \le i \le 31 : f[i-1]\ \equiv\ i \in s \,\rangle\ . \end{aligned}$$

To establish the final conjunct of the postcondition, we try an assignment $a, f := a \cup \{n\}, f'$ for some f'. We now construct f' in such a way that I is reestablished:

$$\begin{array}{ll}
|[\ \ M \wedge I \wedge 1 \le n \le 31 \wedge a = a_\bullet & \\
\triangleright & \\
\quad (a, f := a \cup \{n\}, f').I & \\
\equiv & \{\text{substitution}\} \\
\quad \langle\, \forall i :: f'[i{-}1]\ \equiv\ i \in a \cup \{n\} \,\rangle & \\
\equiv & \{\text{set calculus}\} \\
\quad \langle\, \forall i :: f'[i{-}1]\ \equiv\ i \in a \vee i = n \,\rangle & \\
\equiv & \{\ I\ \} \\
\quad \langle\, \forall i :: f'[i{-}1]\ \equiv\ f[i{-}1] \vee i = n \,\rangle & \\
\equiv & \{\text{domain split, using } 1 \le n \le 31\ \} \\
\quad \langle\, \forall i : i \ne n : f'[i{-}1] \equiv f[i{-}1] \,\rangle\ \wedge\ f'[n{-}1] & \\
\equiv & \{\text{array notation, cf. [2]}\} \\
\quad f' = (f; n{-}1 : \mathit{true}) & \\
]|\ \ , &
\end{array}$$

so after elimination of a we have found the implementation

$$f[n{-}1] := \mathit{true}\ \ .$$

6 Subtypes

Interfaces may be extended, producing a *subtype*. The fact that calling a method specified in an interface may, by dynamic binding, result in the execution of a method implementation corresponding to some subtype, puts constraints on the allowable specifications in the subtype. Consider an interface C with model variable a and model invariant M. Let $C0$ be a subtype of C, with model variable b and model invariant $M0$. Let S be an implementation of a method of $C0$, according to implementation invariant $I0$. Let $U0$ be the precondition and $V0$ the postcondition from that method's specification; it is assumed that $U0$ includes a term $b = b_\bullet$. According to the discussion in the previous section, this is to be interpreted as

$$\langle\, \forall b_\bullet :: \{(b := b_\bullet).(M0 \wedge I0 \wedge U0)\} \;\; S \;\; \{\langle\, \exists b :: M0 \wedge I0 \wedge V0 \,\rangle\}\,\rangle\ . \tag{7}$$

From the preceding Hoare triple we need to conclude that S is also a valid implementation of the corresponding specification in C, say with precondition U and postcondition V. This is the case if

$$\langle\, \forall a_\bullet :: \{(a := a_\bullet).(M \wedge I \wedge U)\} \;\; S \;\; \{\langle\, \exists a :: M \wedge I \wedge V \,\rangle\}\,\rangle \tag{8}$$

for some implementation invariant I, to be determined next.

Led by the shape of the formulae, we decide on the choice

$$I: \quad \langle\, \exists b :: M0 \wedge I0 \wedge R \,\rangle \tag{9}$$

for some R to be determined next. Predicate R forms the connection between a and b and is called the *subtype invariant*. Assume that (7) is given and consider a concrete state and a value $a_\bullet$ for which the precondition of (8) holds. With $X_\bullet$, for X a predicate, an abbreviation for $a, b := a_\bullet, b_\bullet).X$, this precondition may be rewritten as follows:

$$\begin{array}{ll}
 & (a := a_\bullet).(M \wedge I \wedge U) \\
\equiv & \quad \{(9)\} \\
 & (a := a_\bullet).(M \wedge \langle\, \exists b :: M0 \wedge I0 \wedge R \,\rangle \wedge U) \\
\equiv & \quad \{\text{distribution}\} \\
 & \langle\, \exists b :: M0 \wedge I0 \wedge (a := a_\bullet).(M \wedge R \wedge U) \,\rangle \\
\equiv & \quad \{\, \bullet \text{ provided } [M \wedge R \wedge U \;\Rightarrow\; U0] \,\} \\
 & \langle\, \exists b :: M0 \wedge I0 \wedge (a := a_\bullet).U0 \wedge (a := a_\bullet).(M \wedge R \wedge U) \,\rangle \\
\equiv & \quad \{\, U0 \text{ is independent of } a \,\} \\
 & \langle\, \exists b :: M0 \wedge I0 \wedge U0 \wedge (a := a_\bullet).(M \wedge R \wedge U) \,\rangle \\
\equiv & \quad \{\text{dummy transformation}\} \\
 & \langle\, \exists b_\bullet :: (b := b_\bullet).(M0 \wedge I0 \wedge U0 \wedge (a := a_\bullet).(M \wedge R \wedge U)) \,\rangle \\
\equiv & \quad \{\, M0, I0, U0 \text{ independent of } a\,; \text{distribution}\} \\
 & \langle\, \exists b_\bullet :: (M0 \wedge I0 \wedge U0 \wedge M \wedge R \wedge U)_\bullet \,\rangle\ .
\end{array}$$

Choose $b_\bullet$ to be a witness for the term of the existential quantification in the last line. That is the $b_\bullet$ for which we now apply (7) and for which we can establish the postcondition

$$\langle\, \exists b :: M0 \wedge I0 \wedge V0 \,\rangle\ .$$

This can be transformed into the desired postcondition from (8) as follows:

$$
\begin{array}{ll}
|[& (M0 \wedge I0 \wedge U0 \wedge M \wedge R \wedge U)_\bullet \\
\triangleright & \\
 & \langle\, \exists b :: M0 \wedge I0 \wedge V0 \,\rangle \\
\Rightarrow & \quad \{\, \bullet \text{ provided } [V0 \Rightarrow \langle\, \exists a :: M \wedge R \wedge V \,\rangle] \,\} \\
 & \langle\, \exists b :: M0 \wedge I0 \wedge \langle\, \exists a :: M \wedge R \wedge V \,\rangle \,\rangle \\
\equiv & \quad \{\text{distribution}\} \\
 & \langle\, \exists a :: M \wedge \langle\, \exists b :: M0 \wedge I0 \wedge R \,\rangle \wedge V \,\rangle \\
\equiv & \quad \{(9)\} \\
 & \langle\, \exists a :: M \wedge I \wedge V \,\rangle \\
]| & ,
\end{array}
$$

so we have shown that the required postcondition of (8) is indeed established provided

$$[M \wedge R \wedge U \;\Rightarrow\; U0] \;, \tag{10}$$

$$[(M0 \wedge I0 \wedge U0 \wedge M \wedge R \wedge U)_\bullet \;\wedge\; V0 \;\Rightarrow\; \langle\, \exists a :: M \wedge R \wedge V \,\rangle] \;. \tag{11}$$

By way of example, we may extend the interface *DateSet* introduced earlier to a subtype *DateBag* specified through the model

$$b : \mathbb{Z} \to \mathbb{N}$$

with model invariant

$$M0 : \langle\, \forall i : b.i > 0 : 1 \le i \le 31 \,\rangle \;.$$

Its method *add* has a specification with modification clause b, precondition $1 \le n \le 31$, and postcondition $b = (b_\bullet;\; n : b_\bullet.n + 1)$. In order to verify that *DateBag* is correctly identified as a subtype of *DateSet*, we choose the subtype invariant

$$R : \quad \langle\, \forall i :: b.i > 0 \;\equiv\; i \in a \,\rangle \;.$$

Here are the details of the verification of the specification of *add*. Condition (10) is trivially satisfied as U and $U0$ coincide. As to (11), we calculate

$$
\begin{array}{ll}
|[& (M0 \wedge I0 \wedge U0 \wedge M \wedge R \wedge U)_\bullet \wedge V0 \\
\triangleright & \\
 & M \wedge R \wedge V \\
\equiv & \quad \{\text{substitute definitions}\} \\
 & \langle\, \forall i : i \in a : 1 \le i \le 31 \,\rangle \;\wedge\; \langle\, \forall i :: b.i > 0 \;\equiv\; i \in a \,\rangle \;\wedge\; a = a_\bullet \cup \{n\} \\
\equiv & \quad \{\, \bullet \text{ choose } a \text{ as } a_\bullet \cup \{n\} \,\} \\
 & \langle\, \forall i : a \in a_\bullet \cup \{n\} : 1 \le i \le 31 \,\rangle \;\wedge\; \langle\, \forall i :: i \in a_\bullet \cup \{n\} \;\equiv\; b.i > 0 \,\rangle \\
\equiv & \quad \{\text{split off } i = n \,\} \\
 & \langle\, \forall i : i \in a_\bullet : 1 \le i \le 31 \,\rangle \;\wedge\; 1 \le n \le 31 \;\wedge\; \langle\, \forall i : i \ne n : i \in a_\bullet \;\equiv\; b.i > 0 \,\rangle \;\wedge\; b.n > 0 \\
\equiv & \quad \{\, M_\bullet \| U_\bullet \,\} \\
 & \langle\, \forall i : i \ne n : i \in a_\bullet \;\equiv\; b.i > 0 \,\rangle \;\wedge\; b.n > 0 \\
\equiv & \quad \{\, V0 \,\} \\
 & \langle\, \forall i : i \ne n : i \in a_\bullet \;\equiv\; (b; n : b_\bullet.n{+}1).i > 0 \,\rangle \;\wedge\; (b; n : b_\bullet.n{+}1).n > 0 \\
\equiv & \quad \{\text{definition of } (b; n : b_\bullet.n{+}1) \,\} \\
 & \langle\, \forall i : i \ne n : i \in a_\bullet \;\equiv\; b_\bullet.i > 0 \,\rangle \;\wedge\; b_\bullet.n + 1 > 0 \\
\equiv & \quad \{\, R_\bullet \,\}
\end{array}
$$

$$
\begin{array}{ll}
 & b_\bullet.n + 1 > 0 \\
\equiv & \quad \{\, b_\bullet \in \mathbb{Z} \to \mathbb{N} \,\} \\
 & \mathit{true} \\
]| & .
\end{array}
$$

Similar calculations have to be made for the other methods provided. If *DateSet* has a method *delete* with postcondition $a = a_\bullet \setminus \{n\}$, the verification presents no problem in case the corresponding method of *DateBag* has postcondition

$$b = (b_\bullet;\ n : 0)\ ,$$

i.e., all occurrences of n are removed from the bag. However, observe that the postcondition of *delete* could reasonably have been replaced by

$$b = (b_\bullet; n : (b_\bullet.n - 1)\ \uparrow\ 0)\ ,$$

where $\uparrow$ is the infix maximum operator, i.e., a single occurrence of n is removed from the bag.. In that case *DateBag* would *not* have been a correct subtype of *DateSet* .

7 Vulnerability

In the preceding sections, the various kinds of invariants we encountered occurred only at the beginning and end of method bodies. This is, in fact, sufficient if it can be guaranteed that no two method calls on the same object can be ongoing simultaneously. If this guarantee cannot be upheld, *reentrancy* may result and an object may start executing a message body in an inconsistent intermediate state. A way to strengthen the proof obligations to allow for this possibility is shown in [7].

References

[1] M. Abadi and K.R.M. Leino, *A logic of object-oriented programs*. SRC Research Report 161. Digital Systems Research Center, Palo Alto, California, 1998.

[2] E.W. Dijkstra and W.H.J. Feijen, *Een methode van programmeren*. Academic Service, The Hague, 1984.

[3] E.W. Dijkstra and C.S. Scholten, *Predicate calculus and program semantics*. Springer, New York, 1990.

[4] A.J.M. van Gasteren and A. Bijlsma, 'An extension of the program derivation format', in: D. Gries and W.-P. de Roever (eds.), *Programming Concepts and Methods* (PROCOMET'98). Chapman and Hall, London, 1998; pp. 167–185.

[5] J.A. Goguen and G. Malcolm, *A hidden agenda*. Technical Report CS97-538, Department of Computer Science and Engineering, University of California at San Diego, 1997. To appear in *Theoretical Computer Science*.

[6] H.B.M. Jonkers, 'Upgrading the pre- and postcondition technique', in: S. Prehn and W.J. Toetenel (eds.), *Formal Software Development Methods* (VDM'91). Lect. Notes Comput. Sci. 551. Springer, Berlin, 1991; pp. 428–456.

[7] K. Huizing, R. Kuiper, and SOOP, 'Verification of object oriented programs using class invariants', in: T. Maibaum (ed.), *Fundamental Aspects of Software Engineering* (FASE2000). Lect. Notes Comput. Sci. 1783. Springer, Berlin, 2000; pp. 208–221.

[8] B. Liskov and J. Wing, 'A behavioral notion of subtyping'. *ACM Trans. Prog. Lang. Syst.* **16** (1994), 1811–1841.

[9] B. Meyer, *Object-oriented software construction*, 2nd ed. Prentice-Hall PTR, Upper Saddle River, New Jersey, 1997.

[10] C. Morgan and P.H.B. Gardiner, 'Data refinement by calculation'. *Acta Inf.* **27** (1990), 481–503.

[11] J.M. Morris, 'Laws of data refinement'. *Acta Inf.* **26** (1989), 287–308.

[12] H. Mössenböck and N. Wirth, 'The programming language Oberon-2'. *Struct. Prog.* **12** (1991), 179–195.

[13] J. Warmer and A. Kleppe, *The Object Constraint Language*. Addison-Wesley, Reading, Massachusetts, 1999.