The Role of Secondary Attributes in Formal Object Modelling

Jin Song Dong Gordon Rose Roger Duke
Software Verification Research Centre
Department of Computer Science
University of Queensland, Australia
jsdong@cs.uq.oz.au rose@cs.uq.oz.au rduke@cs.uq.oz.au

Abstract

When modelling a large and complex system, clarity of the specification becomes an important factor. In object-oriented specification, the states of individual objects are captured by the values of their attributes. Frequently however, there are dependencies between the attributes of an object. An appropriate indication of which attributes are primary (independent) and which are secondary (dependent) can add significantly to clarity. This paper details the notion of secondary attributes, their roles and implications in formal object-oriented specification.

Keywords: formal specification techniques, object-oriented modelling, object sharing, recursive structures

1 Introduction

When specifying an object, its state is captured by the values of its attributes and its behaviour is implied by its operations. In general, there are many ways to model an object. For example, the shape of a triangle (Figure 1) can be modelled as an object with three attributes representing either (the lengths of) the three sides of the triangle, or two sides and (the size of) their included angle, or two angles and the side between the two angles. These three alternatives are specified in three skeletal classes, $Triangle_{SSS}$, $Triangle_{SAS}$ and $Triangle_{ASA}$ using the Object-Z specification language[**?**, **?**, **?**].

$Triangle_{SSS}$

$a, b, c : \mathbb{R}^+$

$Triangle_{SAS}$

$a, B, c : \mathbb{R}^+$

$Triangle_{ASA}$

$A, b, C : \mathbb{R}^+$

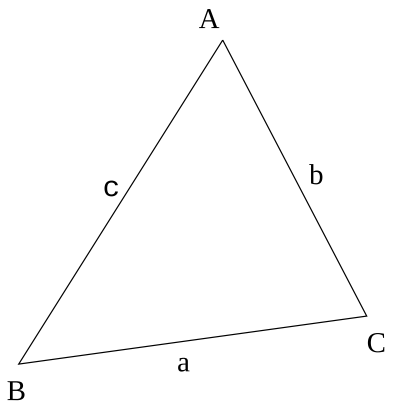

Figure 1: Triangle ABC.

$\mathbb{R}^+$ represents the positive, non-zero real numbers. The three representations all use the minimum number of attributes to uniquely capture the shape of a triangle. However preference depends on context. If all conventional aspects of a triangle need to be represented then a triangle would be modelled as follows:

Triangle

$a, b, c, A, B, C, area, perimeter : \mathbb{R}^+$

Clearly, there are many dependencies between these eight attributes,
e.g. $perimeter = a + b + c$ and $\frac{a}{sin(A)} = \frac{b}{sin(B)} = \frac{c}{sin(C)}$. In general, it is debatable which attributes are more important. However, given a specific system, there is normally an implied preference and an explicit indication of this preference can add clarity to the specification. In large and complex systems, clarity becomes particularly important. Rumbaugh[**?**] suggests that the contents of an object should be separated into two parts — *base* (primary) information and *derived* (secondary) information. For instance, if the

sides of a triangle are given preference, then attributes a, b, c of *Triangle* become the base attributes and other attributes become derived. The notion of secondary attributes has also been realised in [?], where a similar concept to Rumbaugh's *derived attribute*, namely *dependent variable*, is introduced with respect to Object-Z. Although Rumbaugh's introduction of the concept of derived information is informal, it provides a good starting point for discussion of the notion of secondary attributes in Object-Z. This paper formalises and extends the notion of secondary attributes in object-oriented formal modelling. The structure of the paper is as follows.

Section 2 discusses the definition of secondary attributes using a simple hotel management system as an example. Section 3 illustrates how secondary attributes can simplify and clarify a specification. Section 4 shows the use of a secondary attribute in modelling object sharing, and demonstrates that the value of a secondary attribute may depend on the environment rather than on the primary attributes of the object itself. Section 5 presents the role of the secondary attributes in modelling recursively structured objects.

2 Secondary Attributes and Delta Lists

Consider a simple hotel management system that stores information on rooms and their occupancy. A client may *CheckIn* to an unoccupied room or *CheckOut* of an occupied room. Vacant rooms can be listed at any time (*ListVacancies*). Let [*Room*] represent the type of a room in a hotel. This simple system can be modelled in Object-Z as:

HotelManagementSystem

$rooms : \mathbb{P}\, Room$
$occupied : \mathbb{P}\, Room$
Δ
$vacant : \mathbb{P}\, Room$

$occupied \subseteq rooms$
$vacant = rooms - occupied$

CheckIn

$\Delta(occupied)$
$r? : Room$

$r? \in vacant$
$occupied' = occupied \cup \{r?\}$

CheckOut

$\Delta(occupied)$
$r? : Room$

$r? \in occupied$
$occupied' = occupied \setminus \{r?\}$

ListVacancies

$rooms! : \mathbb{P}\, Room$

$rooms! = vacant$

The primary attributes are *rooms* and *occupied* while the only secondary attribute is *vacant*. The dependency of the secondary attribute *vacant* on primary attributes *rooms* and *occupied* is specified in the second conjunct of the class invariant. In Object-Z, the declaration of primary and secondary attributes are syntactically separated by the Δ symbol, which indicates that secondary attributes are implicitly included in the Δ-list of every operation. The understanding of the Δ-list of an operation in Object-Z is that attributes which are so listed are subject to change. Therefore secondary attributes are always subject to change whenever any operation is invoked. Primary attributes not included in the Δ-list of an operation are implicitly unchanged on application of the operation. For instance, if operation *CheckIn* is applied, the predicate $rooms' = rooms$ is implicitly included, whereas the other attributes, *occupied* and *vacant*, are subject to change. The changes are specified by the predicate of *CheckIn* and the class invariant. Note that even though the secondary attribute *vacant* is implicitly included in the Δ-list of operation *ListVacancies*, it remains unchanged because both attributes, *rooms* and *occupied*, being unlisted are unchanged and *vacant* is functionally determined by them.

The hotel management example has illustrated the underlying semantics of a secondary attribute in terms of the Δ-list. In the following sections, the different roles of secondary attributes in formal system modelling are demonstrated.

3 Adding Clarity in Specifications

Consider a complex variable with two component attributes, a real part and an imaginary part, and suppose the variable can be changed by the following operations (Figure 2):

(1) add a given real number to its real part,

(2) add a given real number to its imaginary part,

(3) rotate by a given angle and

(4) extend the modulus by a given positive real number.

When modelling a complex variable as an object, if only two attributes, *real* and *imag*, are considered, then operations *Rotate* and *Extend* (particularly *Rotate*) are difficult and cumbersome to construct. However, if additional secondary attributes, such as the modulus and the argument, are introduced, then operations *Rotate* and *Extend* can be easily defined. A suitable model of a complex variable in Object-Z is the class *Complex*.

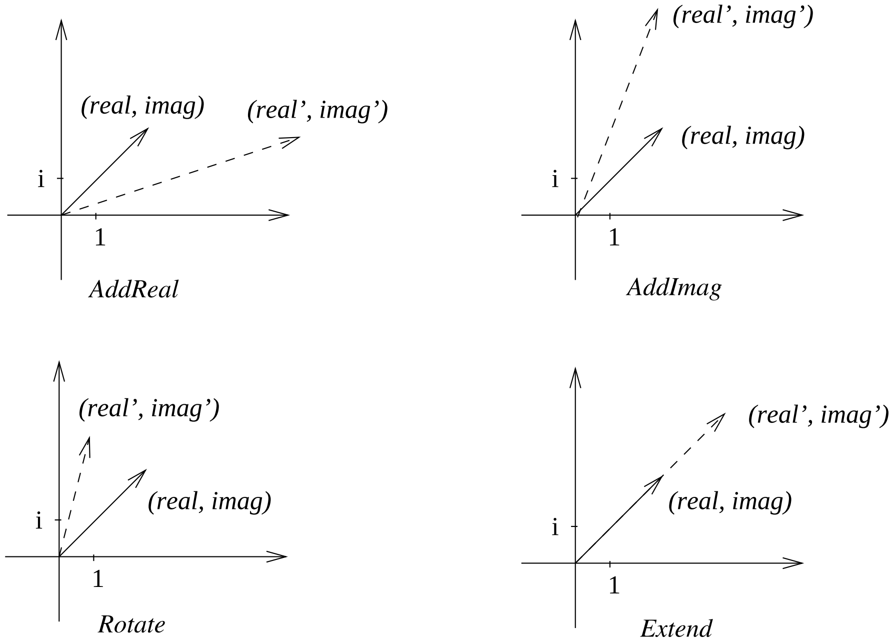

Figure 2: Operations to manipulate a complex number variable.

Complex

$real, imag : \mathbb{R}$
Δ
$modulus, argument : \mathbb{R}$

$modulus = \sqrt{real^2 + imag^2}$
$modulus * sin(argument) = imag$
$modulus * cos(argument) = real$

AddReal

$\Delta(real)$
$r? : \mathbb{R}$

$real' = real + r?$

AddImag

$\Delta(imag)$
$i? : \mathbb{R}$

$imag' = imag + i?$

Rotate

$\Delta(real, imag)$
$\alpha? : \mathbb{R}$

$argument' = argument + \alpha?$
$modulus' = modulus$

Extend

$\Delta(real, imag)$
$l? : \mathbb{R}^+$

$modulus' = modulus + l?$
$argument' = argument$

The dependency of the secondary attributes (*modulus*, *argument*) on primary attributes (*real*, *imag*) is specified in the class invariant.

The benefit of introducing the secondary attributes *modulus* and *argument* in *Complex* is that operations *Rotate* and *Extend* are easily defined by expressing change in terms of them; moreover, in this case the required changes to the primary attributes *real* and *imag* are unambiguously deducible. Another interesting point demonstrated by this example is that, in a particular situation, the value of a secondary attribute need not be uniquely determined. For instance, when $modulus = 0$, the value of the secondary attribute *argument* is arbitrary and even if $modulus \neq 0$, *argument* is only determined modulo 2π.

In the next section, we consider a system in which the value of the secondary attribute is not determined by its objects' primary attributes but by the environment.

4 Dependency on Environment: Object Sharing

Consider a situation in which a group of personal computer (PC) users share their computer games through a communication device. Each member of the group owns a PC and a distinct set of computer games stored in the PC. Individual users can add a new game to or delete an existing game from their PC. Any game of any user can be accessed (played) by all users of the group. Games can also be transferred between users. See Figure 3 for an illustration. The following is an Object-Z specification of this system.

Firstly, let skeletal class *Game* represent a computer game.

Game

$\ldots$

Play

$\ldots$

A user of the group can be modelled as:

User

$my\_games : \mathbb{P}\, Game_{©}$
Δ
$other\_games : \mathbb{P}\, Game$

AddGame

$\Delta(my\_games)$
$g? : Game$

$g? \notin my\_games$
$my\_games' = my\_games \cup \{g?\}$

DelGame

$\Delta(my\_games)$
$g? : Game$

$g? \in my\_games$
$my\_games' = my\_games \setminus \{g?\}$

Choose

$g? : Game$

$g? \in (my\_games \cup other\_games)$

$Play \mathrel{\widehat=} Choose \bullet g?.Play$

The primary attribute *my_games* denotes the local games. The containment notation '©'[?] ensures that each user's *my_games* is a distinct set of games. That is, the following property

$$\forall u_1, u_2 : User \bullet u_1 \neq u_2 \Rightarrow u_1.my\_games \cap u_2.my\_games = \varnothing$$

is implied by the containment notation '©'.

The secondary attribute *other_games* denotes all other public games, i.e. games not local to a user but which can be accessed by the user. However such an intention cannot be expressed as an invariant of the class *User* because, for any user object u, the value of $u.other\_games$ depends on the games which are owned by users other than u in the group, i.e. the value of $u.other\_games$ is dependent on the environment of u. The precise meaning of this attribute is defined by the state invariant of the system class below.

GameSharingGroup

$users : \mathbb{P}\, User$
Δ
$games : \mathbb{P}\, Game$

$games = \bigcup\{u : users \bullet u.my\_games\}$
$\forall u : users \bullet u.other\_games = games \setminus u.my\_games$

Select

$u? : User$

$u? \in users$

SelectTwo

$u_1?, u_2? : User$

$u_1? \neq u_2?$
$\{u_1?, u_2?\} \subseteq users$

$Play \mathrel{\widehat=} Select \bullet u?.Play$
$DiscardGame \mathrel{\widehat=} Select \bullet u?.DelGame$
$NewGame \mathrel{\widehat=} Select \bullet u?.AddGame$
$Transfer \mathrel{\widehat=} SelectTwo \bullet u_1?.DelGame \wedge u_2?.AddGame$

The system consists of a set of users. The class invariant ensures that the value of *other_games* of a user in a group depends on those games which are locally contained by any other user of the group. Incidentally, the definition is facilitated by the introduction of the secondary attribute *games*. This example demonstrates that, given an object $u : User$ in the group, the value of $u.other\_games$ is determined by the existence of some game objects in the environment and those game objects are not even referred to by the primary attribute $u.my\_games$. Therefore the value $u.other\_games$ is subject to change even without applying an operation to u. To illustrate this in detail, let's consider an instance of *GameSharingGroup* (Figure 3).

Suppose the system performs an operation *DiscardGame* ($Select \bullet u?.DelGame$); if $User2$ is selected (i.e. $u? = User2$) and a game is deleted from $User2.my\_games$, then the game is also deleted from $User1.other\_games$, $User3.other\_games$ and $User4.other\_games$.

Although every game is accessible (for *Play*) by all the users of the group, the existence of local games (in *my_games*) is controlled solely by the user; i.e. for any user u, $u.my\_games$ can be changed only by performing $u.AddGame$ or $u.DiscardGame$, whereas u has no control of $u.other\_games$. In the definition of *User*, this difference is captured precisely by distinguishing *my_games* and *other_games* as primary and secondary respectively.

In the next section, we investigate the role of secondary attributes in modelling recursive structures.

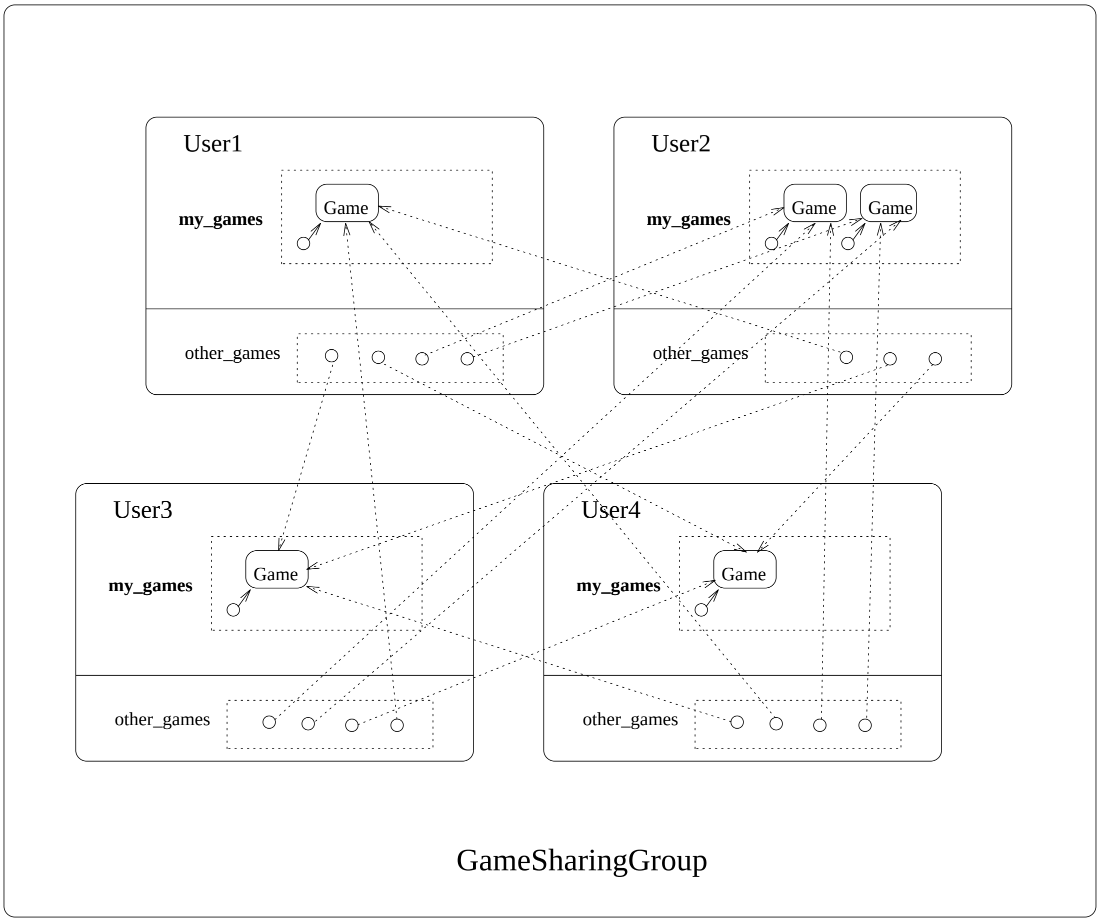

Figure 3: A group of PC users sharing games.

5 Dependency Chains and Recursion

If a component object of a composite object has the same type as the composite object, then the composite object is recursively defined. Frequently, a property of a composite object depends on its component objects. Such dependencies require that the object reference chain from the composite object through its component objects be acyclic. Secondary attributes can be used to capture both the dependency chain and the acyclic object reference structures in recursive definitions. This situation arises frequently when specifying denotational semantics, and the use of secondary attributes helps to clarify and simplify the specification of the semantic domains.

To facilitate this discussion, consider a store of simple predicates. A simple predicate can be a logical variable or a conjunction or a negation. The value of a simple predicate can be output from the store and the value of a logical variable can be updated. Notice that the value of a logical variable is the value stored in the variable while the value of a conjunction depends on its left and right components. If the value of a logical variable is changed then the values of all predicates involved with the logical variable are affected.

All predicates in the store are within the same scope. Figure 4 illustrates the syntactic structure of two predicates '`(A and B) and C`' and '`not (B and C)`' which both refer to the same logical variables '`B`' and '`C`'. To model these predicates in Object-Z, three classes model the three kinds of predicate, namely logical variables, conjunctions and negations. The three classes have a common property — a logical value. This common part is modelled as a class *BasePredicate* which is inherited in the definition of the three predicate classes — *LogicVar*, *Conjunction* and *Negation*:

BasePredicate

Δ
$value : \mathbb{B}$
$subs : \mathbb{P}\, Predicate$

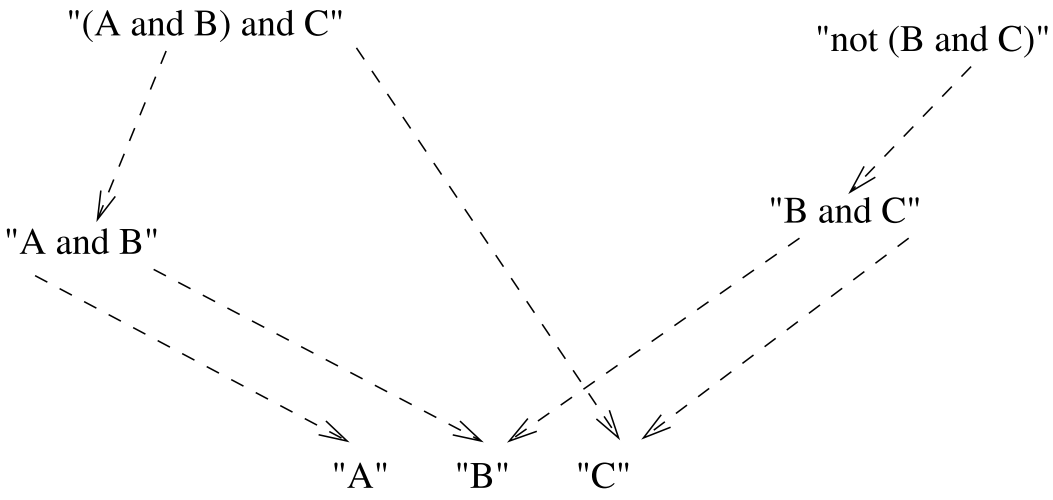

Figure 4: Syntax Structures of Two Predicates.

where *Predicate* is a class-union[**?**, **?**]:

$$Predicate \mathrel{\widehat{=}} LogicVar \cup Conjunction \cup Negation$$

which includes any type of a predicate.

The secondary attribute *value* represents the value of a predicate. Secondary attribute *subs* represents the collection of direct and indirect component object identities: it will facilitate the specification of an acyclic reference structure of any predicate.

A logical variable (an object of class *LogicVar*) has a *name* and a *content*, a boolean value, which can be changed by operation *Change*. *LogicVar* is derived by inheritance from *BasePredicate*.

$$\begin{array}{l}
\textit{LogicVar} \\
\hline
\textit{BasePredicate} \\
\quad \begin{array}{|l}
name : Id \\
content : \mathbb{B} \\
\hline
value = content \\
subs = \varnothing
\end{array} \\
\quad \begin{array}{|l}
\textit{Change} \\
\hline
\Delta(content) \\
value? : \mathbb{B} \\
\hline
content' = value?
\end{array}
\end{array}$$

The class invariant '$value = content$' indicates that the value of a variable is that stored in the variable, while '$subs = \varnothing$' indicates that a logical variable has no sub-component object.

A conjunction predicate consists of a left and a right hand predicate and a negation predicate consists of the predicate which is negated. They are also derived by inheriting *BasePredicate*.

$$\begin{array}{l}
\textit{Conjunction} \\
\hline
\textit{BasePredicate} \\
\quad \begin{array}{|l}
l, r : Predicate \\
\hline
value = (l.value \land r.value) \\
subs = \{l, r\} \cup l.subs \cup r.subs \\
self \notin subs
\end{array}
\end{array}$$

$$\begin{array}{l}
\textit{Negation} \\
\hline
\textit{BasePredicate} \\
\quad \begin{array}{|l}
p : Predicate \\
\hline
value = \neg\, p.value \\
subs = \{p\} \cup p.subs \\
self \notin subs
\end{array}
\end{array}$$

In the definition of *Conjunction* and *Negation*, the class invariants '$value = (l.value \land r.value)$' and '$value = \neg\, p.value$' ensure that the value of a conjunction or a negation depend on the value of their components and ultimately on the primary attributes (*content*) of the variables. The class invariants involved with the secondary attributes *subs* capture the acyclic property of the object reference structure. As an illustration, Figure 5 shows the state and the object reference structures of the two predicates '`(A and B) and C`' and '`not (B and C)`' of Figure 4. The dependency chain from the *values* of '`(A and B) and C`' and of '`not (B and C)`' to the *values* of the variables '`A`', '`B`' and '`C`' are illustrated; for instance, if the *content* of the variable '`C`' is changed from '`false`' to '`true`', the value of the predicates '`(A and B) and C`' and '`not (B and C)`' will be negated.

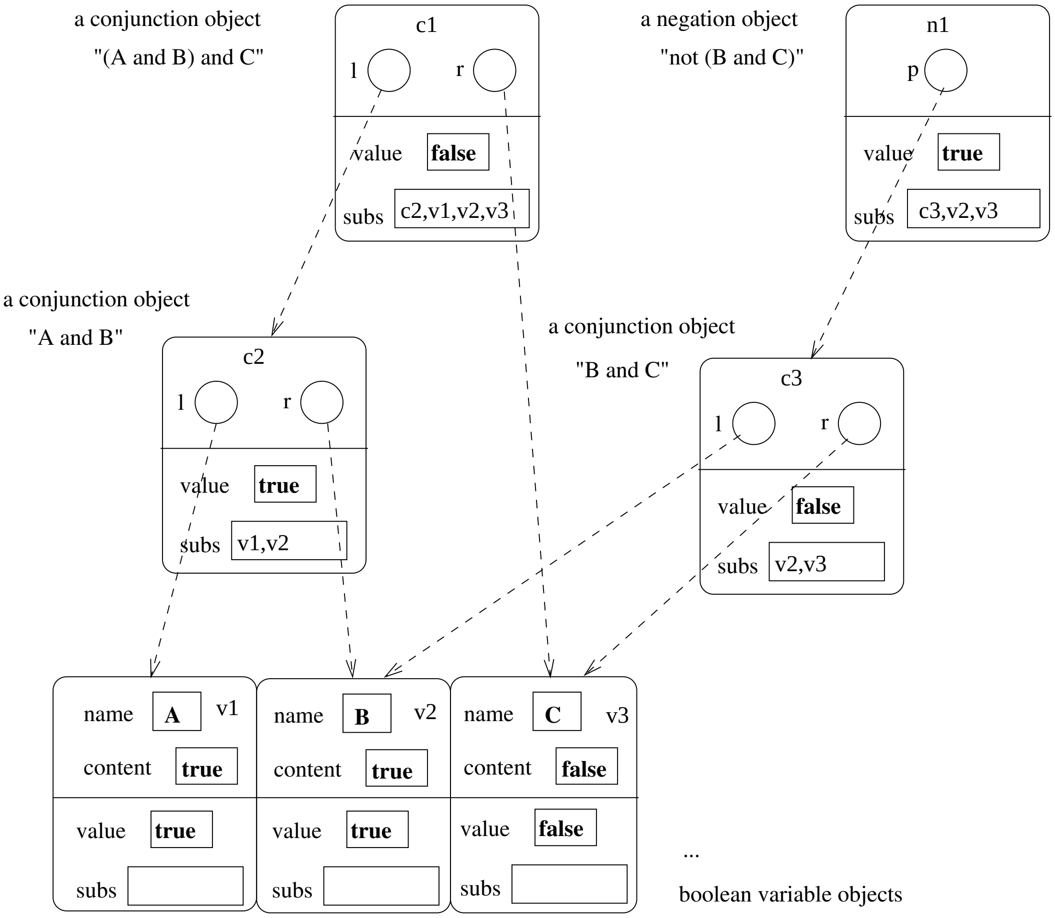

Figure 5: Object Structures of the Two Predicates.

A store of predicates can be modelled as

PredStore

$$
\begin{array}{l}
preds : \mathbb{P}\ Predicate \\
\Delta \\
vars : \mathbb{P}\ LogicVar \\
\hline
vars = \{ v : LogicVar \mid \\
\qquad v \in (\bigcup \{ p : preds \bullet p.subs \} \cup preds) \} \\
\# \{ v : vars \bullet v.name \} = \# vars
\end{array}
$$

Output

$$
\begin{array}{l}
pred? : Predicate \\
value! : \mathbb{B} \\
\hline
pred? \in preds \wedge value! = pred?.value
\end{array}
$$

$$Change \mathrel{\widehat=} [var? : vars] \bullet var?.Change$$

The secondary attribute *vars* gathers all occurrences (direct and indirect) of variables in the store. The class invariant $\#\{v : vars \bullet v.name\} = \#vars$ ensures that all variables in the store have distinct names so that all predicates in the store are within the same scope. For instance, in Figure 5, '`(A and B) and C`' and '`not (B and C)`' refer to the same variables '`B`' and '`C`'.

The operation *Output* demonstrates that a predicate can be selected from the store and that it's value can be broadcast to the environment of the store. The operation *Change* demonstrates that a boolean variable can be selected and its value updated.

In this section, the importance of using secondary attributes to model recursive structures has been demonstrated.

6 Conclusion

In this paper, the implications of applying secondary attributes in formal specification have been explored. Three small case studies analyse and demonstrate three different usages of secondary attributes in formal modelling, namely:

- improving the clarity and simplicity of object-oriented system specifications in general;
- capturing a notion of object sharing; and
- constructing recursive definitions.

We have seen in this paper how the notion of secondary attribute enables invariant relationships between objects to be clearly specified. This is particularly valuable when designing large or complex computer systems.

The game sharing case study in Section 4 demonstrates that the notions of information distribution and information sharing are facilitated by the use of secondary attributes. We believe that secondary attributes will contribute significantly to the formal modelling of distributed systems. The ideas for constructing recursive definitions in Section 5 have been successfully applied to specify the denotational semantics of programming languages in Object-Z[**?**, **?**].

In this paper, the notion of secondary attribute has been discussed within the context of formal specification. A possible way to implement secondary attributes is to use lookup (remote) functions in programming languages. As formality becomes important in information system design, one possible area for further research is to investigate the relationship between the notion of secondary attribute and the notion of normalisation and functional dependency in relational and object-oriented database design.

Acknowledgements

We would like to thank Steven Atkinson, Graeme Smith and Kinh Nguyen for many helpful discussions on the issues raised in this paper. We also wish to thank the anonymous referees for many helpful suggestions.