Chapter 17

LePUS3 in Classical Logic

This chapter describes the relation between specifications in LePUS3 and the more general notion of specification in mathematical logic. We briefly introduce a notion of a first-order language, use it to unpack our notion of *specification*, and present the axioms of class-based programs.

17.1 LePUS3 AND CLASS-Z AS FIRST-ORDER LANGUAGES

Below we use a standard notion of a first-order predicate logic (FOPL) language. The set of *logical symbols* includes the logical connectives $\wedge$ (conjunction), $\vee$ (disjunction), $\Rightarrow$ (implication), $\Leftrightarrow$ (biconditional), the quantifiers $\forall$ (universal), $\exists$ (existential), and the symbols for set $\{\}$, set membership $\in$ and non membership $\notin$, equation $=$, and inequation $\neq$. The set of nonlogical symbols includes the binary superimposition operator $\otimes$ and set subtraction $-$. *Terms* include unary (*Abstract*), binary (*Inherit*), and transitive binary ($Inherit^+$) relation symbols, *constants* (`linkedList`), and *variables* (*factoryMethod*). If t, t_1, t_2 are terms, then 2^t, $t_1 - t_2$, and $t_1 \otimes t_2$ are also terms. If t_1, t_2 are terms, then $t_1 \neq t_2$ and $t_1 \in t_2$ are *well-formed formulas* (wff). Given the wffs φ and ψ and term t; then $\varphi \wedge \psi$, $\varphi \vee \psi$, $\varphi \Rightarrow \psi$, $\forall x \in t \bullet \varphi$, $\exists x \in t \bullet \varphi$ are also wffs. In Table 12 we sketch how to articulate in such a language some of the declarations and formulas of Class-Z.

The semantics of our FOPL language require a *design model*, a straight forward adaptation of our notions of a *finite structure* (§14.1), and an *interpretation function* (§14.3). The term $2^{\underline{relation}}$ is interpreted as the set of all nonempty subsets of tuples in the interpretation of *Relation*. Given a higher dimensional term `T`, the term `T`$-$`t` is interpreted as $\mathcal{I}(\mathtt{T}) - \{\mathcal{I}(\mathtt{t})\}$, that is, the set of entities that is the interpretation of `T` minus the set containing only the interpretation of `t`. Superimposition terms are interpreted as specified in the

Codecharts: Roadmaps and Blueprints for Object-Oriented Programs, by Amnon H. Eden

first part of the book (also formulated in Definition V). Beyond these, Tarski's standard truth conditions for the usual quantifiers and connectives (e.g., [Huth & Ryan 2000]) apply.

Table 21. Unpacking Some LePUS3/Class-Z Declarations and Formulas in the First-Order Predicate Logic

$\mathsf{C}:\mathbb{CLASS}$	$\triangleq$	$\mathsf{C} \in \underline{Class}$
$\mathsf{S}:\mathcal{P}\,\mathbb{SIGNATURE}$	$\triangleq$	$\mathsf{S} \in 2^{\underline{Signature}}$
$Hrc:\mathbb{HIERARCHY}$	$\triangleq$	$Hrc \in 2^{\underline{Class}} \wedge \exists\, root \in Hrc \bullet$ $\forall\, c \in Hrc \bullet (\, c \neq root \Rightarrow \langle\, c, root \rangle \in \underline{Inherit}^{+}\,)$
$Relation(\mathsf{x},\mathsf{y})$	$\triangleq$	$(\langle \mathsf{x},\mathsf{y} \rangle \in \underline{Relation}) \vee$ $(\exists\, sup \bullet Relation(\, sup, \mathsf{y})$ $\wedge\, Inherit(\mathsf{x}, sup)\,) \vee$ $(\exists\, sub \bullet Relation(\mathsf{x}, sub)$ $\wedge\, Inherit(\, sub, \mathsf{y})\,)$
$ALL(\, Unary, X)$	$\triangleq$	$\forall x \in X \bullet x \in \underline{Unary}$
$TOTAL(\, Relation, \mathsf{X}, \mathsf{Y})$	$\triangleq$	$\forall x \in \mathsf{X} \bullet$ $(\, x \in \underline{Method} \wedge x \in \underline{Abstract}) \vee$ $(\exists\, y \in \mathsf{Y} \bullet Relation(\, x, y)\,)$
$ISOMORPHIC(\, Relation, \mathsf{X}, \mathsf{Y})$	$\triangleq$	$\forall x \in \mathsf{X}\ \exists\, y \in \mathsf{Y} \bullet$ $(\, \underline{Relation}(\, x, y) \vee x, y \in \underline{Abstract})\,) \wedge$ $ISOMORPHIC(\, Relation, \mathsf{X} - x, \mathsf{Y} - y)$

Note: Predicates are defined for *1*-dimensional terms; the unpacking of predicates for arguments of higher dimensions is a natural extension

17.2 SPECIFICATIONS IN THE PREDICATE LOGIC

Complete LePUS3 specifications can also be unpacked in the FOPL. To do so we turn to "The Foundations of Specification" [Turner 2005], in which the context is set for a discussion in the mathematical properties of specification languages. Turner develops a core specification theory (CST) and shows that his general account adequately unpacks statements in many formal specification languages. Turner's analysis provides us with a general and well-defined notion of a *specification*, which can be paraphrased as follows. Having unpacked in our FOPL the truth conditions for individual formulas in our specification languages (Table 21), we may now move to unpack complete LePUS3 and Class-Z specifications as sentences in the FOPL.

In Chapter 15 we distinguished between *open specifications* (a specification with variables) and *closed specifications*. We shall employ this distinction in the discussion in the translation of LePUS3 specifications to FOPL. We say that the variable x is *free* in the FOPL wff φ if at least one of its occurrences in φ is not bound by any quantifier within φ. A wff is *closed* if and only if it contains no free variables.

Let Ψ designate a closed specification whose set of declarations and formulas are equivalent (by Table 21) to the set of k closed FOPL wffs $\psi_1, \ldots, \psi_k$. Then Ψ introduces a new predicate Ψ as follows:

$$\Psi \triangleq \psi_1 \wedge \ldots \wedge \psi_k$$

For example, consider the closed schema ItrNextReturnsObject (equivalent to Codechart 25, p. 61):

ItrNextReturnsObject

object, arrayListItr: $\mathbb{CLASS}$ next: $\mathbb{SIGNATURE}$
Return(next $\otimes$ arrayListItr, object)

Schema ItrNextReturnsObject is unpacked in the FOPL to introduce the predicate *ItrNextReturnsObject* as follows:

ItrNextReturnsObject $\triangleq$

object $\in$ *Class* $\wedge$

arrayListItr $\in$ *Class* $\wedge$

next $\in$ *Signature* $\wedge$

$(\,(\,\langle$ next $\otimes$ arrayListItr, object $\rangle \in$ *Return*$)\,\vee$

$(\,\exists\, sub \bullet$ *Return*(next $\otimes$ arrayListItr, *sub*$) \wedge$ *Inherit*(*sub*, object$)\,)\,)$

We may generalize this description to open specifications as follows. Let Ψ designate a LePUS3 specification with n variables $x_1, \ldots, x_n$ where n is a natural number. (When $n = 0$, Ψ is a closed specification). Let $\psi_1[x_1, \ldots, x_n], \ldots, \psi_k[x_1, \ldots, x_n]$ designate the list of FOPL wffs, each of which represents the unpacking of exactly one declaration or Class-Z formula in Ψ as indicated by Table 21, where the set of free variables for each ψ_i is a (possibly empty) subset of $\{x_1, \ldots, x_n\}$. Then Ψ introduces the n-ary predicate symbol $\Psi[x_1, \ldots, x_n]$ into the language as follows:

$$\Psi[x_1, \ldots, x_n] \triangleq \psi_1[x_1, \ldots, x_n] \wedge \ldots \wedge \psi_k[x_1, \ldots, x_n]$$

For example, the open schema CompositeComponent schema (Figure 16-1, p. 194) is unpacked in the FOPL as a definition that introduces the ternary predicate symbol CompositeComponent [*composite*, *component*, *Ops*] defined as follows:

CompositeComponent[*composite*, *component*, *Ops*] $\triangleq$

Composite $\in$ *Class* $\wedge$

$$
\begin{aligned}
&\mathit{Component} \in \underline{\mathit{Class}} \wedge \\
&\mathit{Ops} \in \mathit{Set}(\underline{\mathit{Signature}}) \wedge \\
&(\forall \mathit{orig} \in (\mathit{Ops} \otimes \mathit{composite}) \bullet \\
&(\mathit{orig} \in \underline{\mathit{Method}} \wedge \mathit{orig} \in \underline{\mathit{Abstract}}) \vee \\
&(\exists \mathit{dest} \in (\mathit{Ops} \otimes \mathit{component}) \bullet \langle \mathit{orig}, \mathit{dest} \rangle \in \underline{\mathit{Forward}}))
\end{aligned}
$$

17.3 THE AXIOMS OF CLASS-BASED PROGRAMS

The abstract semantics for LePUS3 were defined in Chapter 14 in terms of *finite structures* and *design models*. Among the conditions imposed on design models in Chapter 14 is the requirement that they satisfy the *axioms of class-based programs* (Definition VIII). What are these axioms and why are they necessary?

By *axiomatization* we refer to the process of articulating formally the constraints imposed on a particular collection of model-theoretic structures. The axioms of class-based programs articulate some of the basic conditions that any design model that appropriately represents a valid program must satisfy by virtue of the properties of class-based programming languages. In other words, these axioms restrict the abstract semantics to exclude many of the models that under no circumstances may appropriately represent a valid program.

Consider, for example, the constraint imposed by all class-based programming languages that a class may not inherit from itself, directly or indirectly. Axiom 2 holds because design models are the abstract semantics *only* for well-formed programs, and because all well-formed programs satisfy this condition. In other words, Axiom 2 captures and makes explicit a constraint that the syntactic and semantic rules of class-based programming languages impose on the manner by which inheritance relations can be defined in well-formed programs.

Ideally, the combination of these axioms would be complete: It would guarantee that every design model that satisfies them appropriately represents (§14.3, p. 175) some program. Unfortunately, they are not. One of the reasons for this is because class-based programming languages vary by the set of constraints they impose.

Axiom 1. No two methods with the same signature are members of the same class:

$$
\begin{aligned}
&\forall \mathit{sig} \in \underline{\mathit{Signature}}\ \forall \mathit{cls} \in \underline{\mathit{Class}}\ \forall \mathit{mth}_1, \mathit{mth}_2 \in \underline{\mathit{Method}} \bullet \\
&\quad (\langle \mathit{sig}, \mathit{mth}_1 \rangle \in \underline{\mathit{SignatureOf}} \wedge \\
&\qquad \langle \mathit{mth}_1, \mathit{cls} \rangle \in \underline{\mathit{Member}} \wedge \\
&\qquad \langle \mathit{sig}, \mathit{mth}_2 \rangle \in \underline{\mathit{SignatureOf}} \wedge \\
&\qquad \langle \mathit{mth}_2, \mathit{cls} \rangle \in \underline{\mathit{Member}}) \\
&\qquad \Rightarrow \mathit{mth}_1 = \mathit{mth}_2
\end{aligned}
$$

The first axiom represents a rule that is enforced by the compilers of every class-based program language, in particular Java, C++, Smalltalk, and C#, on the relation between methods and classes.[1]

Note that we do *not* require that every method is a member of exactly one class. Although this is true in the Java programming language, it is not true for C++, which allows for methods ("global functions") that are not members of any class. This is neither true in CLOS, which supports multiple dispatch [Craig 2000], a mechanism which associates each method with any (fixed) number of classes.

Axiom 2. (Asymmetry of Transitive Closure of Relation *Inherit*). There are no cycles in the inheritance graph:

$$\forall\, cls_1 \in \underline{Class}\ \forall\, cls_2 \in \underline{Class} \bullet$$
$$\langle cls_1, cls_2 \rangle \notin \underline{Inherit}^+ \ \vee \langle cls_2, cls_1 \rangle \notin \underline{Inherit}^+$$

The second axiom requires that a class may not, directly or indirectly, inherit from itself:

Axiom 3. Every method has exactly one signature.

$$\forall\, mth \in \underline{Method}\ \exists\, sig \in \underline{Signature} \bullet \langle sig, mth \rangle \in \underline{SignatureOf} \wedge$$
$$(\forall\, sig_2 \in \underline{Signature} \bullet \langle sig_2, mth \rangle \in \underline{SignatureOf} \Leftrightarrow sig_2 = sig)$$

The third axiom requires that the notion of a signature is unique for each method. Indeed, a method signature is uniquely defined in the program by the method's name and argument types. It would therefore be meaningless to associate a method with more than one signature.

Axiom 4. Certain relations imply other relations:

$$\forall\, mth \in \underline{Method}\ \forall\, cls \in \underline{Class} \bullet$$
$$\langle mth, cls \rangle \in \underline{Produce} \Rightarrow$$
$$\langle mth, cls \rangle \in \underline{Create} \wedge \langle mth, cls \rangle \in \underline{Return}$$
$$\forall\, mth_1, mth_2 \in \underline{Method} \bullet$$
$$\langle mth_1, mth_2 \rangle \in \underline{Forward} \Rightarrow \langle mth_1, mth_2 \rangle \in \underline{Call}$$
$$\forall\, cls_1, cls_2 \in \underline{Class} \bullet$$
$$\langle cls_1, cls_2 \rangle \in \underline{Aggregate} \Rightarrow \langle cls_1, cls_2 \rangle \in \underline{Member}$$

The fourth axiom requires that the notions of some relations fit our intuitive expectations. For example, if one method is said to forward the call to another, we expect it to be true to say that the first method *calls* the second.

[1]Method "overloading" allows several methods with the same "name" to be defined in the same class, as long as their signatures are sufficiently distinct.