LePUS3 and Class-Z Reference Manual

Technical Report CSM-474, ISSN 1744-8050

Print

Amnon H Eden, Epameinondas Gasparis, Jonathan Nicholson
Department of Computer Science, University of Essex, United Kingdom

31 December 2007 (Minor revisions: 8 July 2008, 2 Oct. 2009)

Abstract

This document formally defines the elements in the syntax and the semantics of LePUS3 and the Class-Z specification languages. It was designed to satisfy the rigid requirements of mathematical logic, and it is therefore unsuitable for learning LePUS3 and Class-Z. To learn LePUS3 and Class-Z please see the Tutorial. To learn about the background and motivation see the About page. A legend offering a key to the language's symbols is also available.

Table of contents

Related links:

- **About** LePUS3 and Class-Z

- **Tutorial**: Object-Oriented Modelling with LePUS3 and Class-Z [.pdf][.ppt]
- **Legend**: Key to LePUS3 and Class-Z Symbols [.pdf]
- **Verification** of LePUS3/Class-Z Specifications: Sample models and Abstract Semantics for Java 1.4 [.pdf] [Nicholson et al. 2007]
    - Part I: Abstract Semantics for Java 1.4 Programs
    - Part II: Sample Models

1. Introduction

LePUS3 and Class-Z are object-oriented Design Description Languages, meaning that they are formal specification and modelling languages for object-oriented design which were tailored to allow tool support in software modelling, specification, verification, and visualization.

LePUS3 and Class-Z are the product of the collaborative effort of the authors, based on revisions to the LanguagE for Patterns Uniform Specification—LePUS [Eden 2001]. Class-Z and LePUS3 are equivalent: Every specification in LePUS3 is equivalent to one encoded in Class-Z and vice versa, the difference being that LePUS3 is a visual language (a specification in LePUS3 is called a Codechart) while Class-Z is a symbolic language which borrows the schema notation from the Z specification language [Spivey 1992].

For more information about LePUS3 and Class-Z please visit: www.lepus.org.uk/about.xml.

1.1. The metalanguage

LePUS3 and Class-Z are defined using classical predicate calculus. It is also easy to show that the both LePUS3 and Class-Z are proper subsets of first-order predicate calculus (see proposition 1). We use the standard language of mathematical logic in defining the semantics of LePUS3 and Class-Z, including model theory, predicate calculus, and elementary set theory. Among others, we use the following symbols which carry their usual meanings:

Symbol	Meaning
iff	if and only if
$\triangleq$	Is defined as
$\in$	Set membership
$\cup$	Set union

The Axioms of Class-Based Programs have also been transcribed to formulas in FOPL, using the quantifiers $\forall$ (Forall), $\exists$ (Exists), and the connectives $\wedge$ (And), $\vee$ (Or), $\Rightarrow$ (implies) with their usual meanings.

In addition, we define the following notations to be used in our metalanguage:

Definition I:

Given a set T, the **power set** $\mathcal{P}T$ is the set of all subsets of T.

For example, the power set of the set $\{\underline{a}, \underline{b}\}$ is a set of four elements as follows:
$\mathcal{P}\{\underline{a}, \underline{b}\} = \{\{\}, \{\underline{a}\}, \{\underline{b}\}, \{\underline{a}, \underline{b}\}\}$

Definition II:

Given a binary relation $Relation$, the **transitive closure** of $Relation$, written $Relation^+$, consists of those pairs $\langle \underline{x}, \underline{y} \rangle$ such that either one of the following holds:

1. $\langle \underline{x}, \underline{y} \rangle \in Relation$, or
2. there exists an entity $\underline{z}$ such that $\langle \underline{x}, \underline{z} \rangle \in Relation$ and $\langle \underline{z}, \underline{y} \rangle \in Relation^+$

For example, since class `Vector` does NOT inherit directly from (extends or implements) class `Object`, it would be false to require that

$$\langle \underline{vector}, \underline{object} \rangle \in Inherit$$

However, `Vector` does inherit INDIRECTLY from `Object` directly, so we require that

$$\langle \underline{vector}, \underline{object} \rangle \in Inherit^+$$

If you are unfamiliar with mathematical logic or with these symbols we recommend you consult any introductory book on elementary logic and set theory such as [Huth & Ryan 2000].

2. Semantics

The semantics of LePUS3 and Class-Z consist of an abstract representation of programs in class-based object-oriented programming languages such as Java, C++, and Smalltalk. The picture our semantics provides, a design model, is called the **abstract semantics** of the program. This simplified representation consists of atomic primitives ('entities of dimension 0') and sets thereof (entities of dimension 1), as well as relations between the primitive entities.

2.1. Entities

Entities represent elements of the abstract semantics of a program: classes, methods, method signatures, and sets of these entities.

Each entity has a **dimension**. We shall be primarily concerned with entities of dimension 0 or 1. An **entity of dimension 0** is an atomic primitive (representing either a class, a method, or a method signature in the program), and an **entity of dimension 1** is a non-empty, finite set of entities of dimension 0. More generally, a non-empty, finite set of entities of dimension d is an entity of dimension $d+1$. An entity of dimension $d>0$ is called a **higher-dimensional entity**.

Entity names are written in underlined fixed-size typeface, where lower case names (e.g., $\underline{\texttt{object}}$, $\underline{\texttt{collection}}$) are reserved for entities of dimension 0, and capitalized names (e.g., $\underline{\texttt{Objects}}$, $\underline{\texttt{Collections}}$) are reserved for entities of dimension 1.

Definition III:
A **class of dimension 0** is an atomic primitive in the unary relation $\underline{Class}$. A **class of dimension 1** is a non-empty, finite set of classes of dimension 0.

In the abstract semantics of Java, classes of dimension 0 represent classes, interfaces, and primitive types (e.g. `int`, `char`, etc.) in the program.

For example, $\underline{\texttt{object}}$ is a class of dimension 0 which represents the class `java.lang.Object` in the abstract semantics of Java programs.

For example, $\underline{\texttt{int}}$ is a class of dimension 0 which represents the primitive type `int` in the abstract semantics of Java and C++ programs.

For example, $\underline{\texttt{collection}}$ is a class of dimension 0 which represents the interface `java.util.Collection` in the abstract semantics of Java 1.4.

For example, the set $\{\underline{\texttt{object}}, \underline{\texttt{int}}, \underline{\texttt{collection}}\}$ is a class of dimension 1 in the abstract semantics of a Java 1.4 program.

Also, the unary relation $\underline{Class}$ is itself also a class of dimension 1.

See more classes of dimension 0 in Example 1 in [Nicholson et al. 2007, Part I]

Definition IV:
A class of dimension 1 $\underline{\texttt{Hrc}}$ is also a **hierarchy of dimension 1** iff it satisfies the following conditions:

1. $\underline{\texttt{Hrc}}$ contains at least two classes of dimension 0
2. There exists $\underline{\texttt{root}}$ a class of dimension 0 in $\underline{\texttt{Hrc}}$ such that for any other class $\underline{\texttt{cls}}$ in $\underline{\texttt{Hrc}}$: $\langle \underline{\texttt{cls}}, \underline{\texttt{root}} \rangle \in \underline{Inherit}^+$

A 'hierarchy' is therefore a set of classes which includes one 'root' class such that all other classes inherit (possibly indirectly) from it.

For example, any set of two or more classes in Java which includes `java.lang.Object` is a hierarchy.

See more hierarchies in Example 2 in [Nicholson et al. 2007, Part II].

Definition V:
A **signature of dimension 0** is an atomic primitive in the unary relation $\underline{Signature}$. A **signature of dimension 1** is a non-empty, finite set of signatures of dimension 0.

Signatures are abstractions of method name and argument types.

For example, the abstract semantics of the `java.util` package shall contain one signature of dimension 0: "`size()`" which represents the signature of both methods `ArrayList.size()` and `LinkedList.size()`.

For example, the abstract semantics of the `java.util` package shall contain one signature of dimension 0: "`add(Object)`" which represents the signature of both methods `ArrayList.add(Object)` and `LinkedList.add(Object)`.

For example, if $\underline{\mathrm{size}}$ and $\underline{\mathrm{add}}$ are signatures of dimension 0 then $\{\underline{\mathrm{size}}, \underline{\mathrm{add}}\}$ is a signature of dimension 1.

See more signatures in Java in Example 2 and Example 3 in [Nicholson et al. 2007, Part I].

Unlike methods, signature entities have a dedicated symbols in LePUS3 and Class-Z (e.g., 0-dimensional and 1-dimensional signature constants) whereas method entities have no dedicate symbols for representing them. Instead, we use superimpositions, the advantage is that of being able to represent a large set of methods indirectly using a single signature.

Definition VI:
A **method of dimension 0** is an atomic primitive in the unary relation $\underline{Method}$. A **method of dimension 1** is a non-empty, finite set of methods of dimension 0.

Method entities abstract the procedural units of execution in class-based programming languages: 'methods' in Java and Smalltalk, functions and function members in C++. However Method entities have no dedicated symbol in LePUS3 and Class-Z. Instead, method entities are represented using the superimposition of signature and class symbols, the advantage is that of being able to represent a large set of methods indirectly using a single signature.

See methods of dimension 0 in Java in Example 2 and Example 3 in [Nicholson et al. 2007, Part I].

2.2. Relations

Relations are simply sets of tuples of entities. We distinguish between unary relations and binary relations as follows:

Definition VII:
A **unary relation** is a set of entities of dimension 0.

For example, the unary relation $\underline{Class}$ contains all the classes of dimension 0 in the abstract semantics of a Java 1.4 program, each of which represents a class, interface, or primitive type. Most commonly, the relation $\underline{Class}$ will contain at least the entities $\underline{\mathrm{object}}$ and $\underline{\mathrm{int}}$ in the abstract semantics of any Java program.

See the unary relation $\underline{Class}$ in the abstract semantics of a Java program in Example 1 and

Example 2 in [Nicholson et al. 2007, Part I].

For example, the unary relation $Method$ contains all the methods of dimension 0 in the abstract semantics of a Java or a C++ program, each of which represents a method (in C++: a *function* or a *function member*).

See the unary relation $Method$ in the abstract semantics of a Java program in Example 2 and Example 3 in [Nicholson et al. 2007, Part I].

For example, the unary relation $Abstract$ contains all those classes of dimension 0 and methods of dimension 0 which represent the abstract methods, abstract classes, and the interfaces in the abstract semantics of a Java program.

See the unary relation $Abstract$ in the abstract semantics of a Java program in Example 4 in [Nicholson et al. 2007, Part I].

Definition VIII:
A **binary relation** is a set of pairs of entities of dimension 0.

For example, the binary relation $Inherit$ represents the `extends`, `implements`, and the subtype relations between Java classes and/or interfaces. For instance, $Inherit = \{ \langle \mathtt{collection}, \mathtt{object} \rangle, \langle \mathtt{list}, \mathtt{collection} \rangle \}$ since the Java interface `Collection` is a subtype of class `Object` and the interface `List` implements the interface `Collection`.

See the binary relation $Inherit$ in the abstract semantics of a Java program in Example 5 in [Nicholson et al. 2007, Part I].

For example, the binary relation $Member$ represents the relation between a class containing a field and the class of the contained field in a Java program (in C++: between a class containing a data member and the class/type of the contained member.)

See the binary relation $Member$ in the abstract semantics of a Java program in Example 7 in [Nicholson et al. 2007, Part I].

2.3. Superimpositions

In LePUS3 and Class-Z, methods have no dedicated symbol. Instead, of the form superimposition terms of the form $\mathtt{sig} \otimes \mathtt{cls}$ represent methods by indicating their signature ($\mathtt{sig}$) and and the class in which they are defined ($\mathtt{cls}$). (C++ global functions are represented as methods that are not members of any class, and multiple-dispatch methods in languages such as CLOS can be members of more than one class.)

Definition IX:
A **0-dimensional superimposition** is an expression of the form $\mathtt{sig} \otimes \mathtt{cls}$, where $\mathtt{sig}$ is a signature of dimension 0 and $\mathtt{cls}$ is a class of dimension 0. The binary operator $\otimes$ is a partial functional relation, defined as follows:

- If there exists $\mathtt{mth}$ a method of dimension 0 such that $\langle \mathtt{sig}, \mathtt{mth} \rangle \in SingatureOf$ and

$\langle \underline{\mathtt{mth}}, \underline{\mathtt{cls}} \rangle \in \underline{Member}$ then

$$\underline{\mathtt{sig}} \otimes \underline{\mathtt{cls}} \triangleq \underline{\mathtt{mth}}$$

- Otherwise, if there exists exactly one $\underline{\mathtt{supercls}}$ class of dimension 0 such that $\langle \underline{\mathtt{cls}}, \underline{\mathtt{supercls}} \rangle \in \underline{Inherit}$ for which $\underline{\mathtt{sig}} \otimes \underline{\mathtt{supercls}}$ is defined and $\underline{\mathtt{sig}} \otimes \underline{\mathtt{supercls}} \in \underline{Inheritable}$ then

$$\underline{\mathtt{sig}} \otimes \underline{\mathtt{cls}} \triangleq \underline{\mathtt{sig}} \otimes \underline{\mathtt{supercls}}$$

- Otherwise the term $\underline{\mathtt{sig}} \otimes \underline{\mathtt{cls}}$ is undefined.

Given $\underline{\mathtt{Signatures}} = \{ \underline{\mathtt{s}}_1, \ldots, \underline{\mathtt{s}}_n \}$ a signature of dimension 1 and $\underline{\mathtt{Classes}} = \{ \underline{\mathtt{c}}_1, \ldots, \underline{\mathtt{c}}_k \}$ a class of dimension 1, we also define the following **1-dimensional superimposition** expressions:

$$\underline{\mathtt{sig}} \otimes \underline{\mathtt{Classes}} \triangleq \{ \underline{\mathtt{sig}} \otimes \underline{\mathtt{c}}_1, \ldots, \underline{\mathtt{sig}} \otimes \underline{\mathtt{c}}_k \}$$

$$\underline{\mathtt{Signatures}} \otimes \underline{\mathtt{cls}} \triangleq \{ \underline{\mathtt{s}}_1 \otimes \underline{\mathtt{cls}}, \ldots, \underline{\mathtt{s}}_n \otimes \underline{\mathtt{cls}} \}$$

The method of dimension 1 $\underline{\mathtt{sig}} \otimes \underline{\mathtt{Classes}}$ is called a **clan**.

The method of dimension 1 $\underline{\mathtt{Signatures}} \otimes \underline{\mathtt{cls}}$ is called a **tribe**.

In other words, the superimposition $\underline{\mathtt{sig}} \otimes \underline{\mathtt{cls}}$ selects the unique method with signature $\underline{\mathtt{sig}}$ that is either defined explicitly in class $\underline{\mathtt{cls}}$ or inherited from exactly one other class.

For example, if $\underline{\mathtt{size}}$ represents the signature of the method `ArrayList.size()` and $\underline{\mathtt{arrayList}}$ represents class `ArrayList` then the superimposition $\underline{\mathtt{size}} \otimes \underline{\mathtt{arrayList}}$ represents the method `ArrayList.size()`.

For example, if $\underline{\mathtt{iterator}}$ represents the signature of the method `AbstractSequentialList.iterator()` and $\underline{\mathtt{linkedList}}$ represents class `LinkedList` then the superimposition $\underline{\mathtt{iterator}} \otimes \underline{\mathtt{LinkedList}}$ represents method `AbstractSequentialList.iterator()` which class `LinkedList` inherits from class `AbstractSequentialList`.

For more superimpositions see Example 4 in [Nicholson et al. 2007, Part II].

2.4. Structures

Our notion of semantics is based on finite structures in model theory. A **finite structure** $\mathfrak{F}$ is simply a pair $\langle \mathbb{U}, \mathbb{R} \rangle$ such that $\mathbb{U}$ (also called the universe of $\mathfrak{F}$) is a finite set of primitive entities (in our metalanguage: entities of dimension 0) and $\mathbb{R}$ is a set of relations. We extend the traditional notion of a finite structure with the notion of a design model as follows:

Definition X:

A **design model** is a triple $\mathfrak{M} \triangleq \langle \mathbb{U}_*, \mathbb{R}, \mathcal{I} \rangle$ such that:

- $\mathbb{U}_*$, called the **universe** of $\mathfrak{M}$, is a finite set of entities such that $\mathbb{U}_* \triangleq \mathbb{U}_0 \cup \mathbb{U}_1$ where $\mathbb{U}_0$ is a finite set of entities of dimension 0 and $\mathbb{U}_1$ is a finite set of entities of dimension 1.
- $\mathbb{R}$ is a set of relations, including:
  - The unary relations $\underline{Class}$, $\underline{Method}$, $\underline{Signature}$, $\underline{Abstract}$, and $\underline{Inheritable}$
  - The binary relations $\underline{Inherit}$, $\underline{Member}$, $\underline{Produce}$, $\underline{Call}$, $\underline{Create}$, $\underline{Forward}$, $\underline{Return}$, $\underline{Aggregate}$, and $\underline{SignatureOf}$
- $\mathcal{I}$ is an **interpretation** function which maps some constant terms to entities in $\mathbb{U}_*$ depending on the type of the term:

t is a constant term of type	If defined, $\mathcal{I}(t)$ is a
$\mathbb{CLASS}$	class of dimension 0
$\mathcal{P}\mathbb{CLASS}$	class of dimension 1
$\mathbb{SIGNATURE}$	signature of dimension 0
$\mathcal{P}\mathbb{SIGNATURE}$	signature of dimension 1
$\mathbb{METHOD}$	method of dimension 0
$\mathcal{P}\mathbb{METHOD}$	method of dimension 1
$\mathbb{HIERARCHY}$	hierarchy of dimension 1

For superimposition terms, we define:

$$\mathcal{I}(\tau_1 \otimes \tau_2) \triangleq \mathcal{I}(\tau_1) \otimes \mathcal{I}(\tau_1)$$

If defined then $\mathcal{I}(\ \tau\ )$ is called the interpretation of τ .

- $\mathfrak{M}$ satisfies the Axioms of Class-Based Programs.

Design models supply us with a greatly simplified picture of the program: They abstract the intricate details of the implementation, which are normally buried deep in the source code, and represent them using of primitive entities ('entities of dimension 0') and relations amongst them. In addition, design models supply us with sets of entities ('entities of dimension 1'), which cluster together related classes and related methods.

Our definition of design models extends naturally to include entities of any finite dimension.

> See sample design models for Java programs see documents "Abstract Semantics for Java 1.4 Programs" [Nicholson et al. 2007, Part I] and "Sample Models" [Nicholson et al. 2007, Part II].

Definition XI:

The following are the **Axioms of Class-Based Programs**:

Axiom 1: No two methods with same signature (method name and argument type) are members of

same class:

$$\forall \underline{\mathtt{sig}} \in \underline{Signature} \quad \underline{\mathtt{cls}} \in \underline{Class} \quad \underline{\mathtt{mth}_1}, \underline{\mathtt{mth}_2} \in \underline{Method} \bullet$$
$$\langle \underline{\mathtt{mth}_1}, \underline{\mathtt{cls}} \rangle \in \underline{Member} \land \langle \underline{\mathtt{sig}}, \underline{\mathtt{mth}_1} \rangle \in \underline{SignatureOf} \land$$
$$\langle \underline{\mathtt{mth}_2}, \underline{\mathtt{cls}} \rangle \in \underline{Member} \land \langle \underline{\mathtt{sig}}, \underline{\mathtt{mth}_2} \rangle \in \underline{SignatureOf} \Rightarrow$$
$$\underline{\mathtt{mth}_1} = \underline{\mathtt{mth}_2}$$

Axiom 2: There are no cycles in the inheritance graph:

$$\forall \underline{\mathtt{cls}_1}, \underline{\mathtt{cls}_2} \in \underline{Class} \bullet$$
$$\langle \underline{\mathtt{cls}_1}, \underline{\mathtt{cls}_2} \rangle \notin \underline{Inherit}^+ \lor \langle \underline{\mathtt{cls}_2}, \underline{\mathtt{cls}_1} \rangle \notin \underline{Inherit}^+$$

Axiom 3: Every method has exactly one signature:

$$\forall \underline{\mathtt{mth}} \in \underline{Method} \quad \exists ! \underline{\mathtt{sig}} \in \underline{Signature} \bullet$$
$$\langle \underline{\mathtt{sig}}, \underline{\mathtt{mth}} \rangle \in \underline{SignatureOf}$$

Axiom 4: Some dependencies exist between relations as follows:

$$\forall \underline{\mathtt{mth}} \in \underline{Method} \quad \underline{\mathtt{cls}} \in \underline{Class} \bullet$$
$$\langle \underline{\mathtt{mth}}, \underline{\mathtt{cls}} \rangle \in \underline{Produce} \Rightarrow \langle \underline{\mathtt{mth}}, \underline{\mathtt{cls}} \rangle \in \underline{Create} \land \langle \underline{\mathtt{mth}}, \underline{\mathtt{cls}} \rangle \in \underline{Return}$$

$$\forall \underline{\mathtt{mth}_1}, \underline{\mathtt{mth}_2} \in \underline{Method} \bullet$$
$$\langle \underline{\mathtt{mth}_1}, \underline{\mathtt{mth}_2} \rangle \in \underline{Forward} \Rightarrow \langle \underline{\mathtt{mth}_1}, \underline{\mathtt{mth}_2} \rangle \in \underline{Call}$$

$$\forall \underline{\mathtt{cls}_1}, \underline{\mathtt{cls}_2} \in \underline{Class} \bullet$$
$$\langle \underline{\mathtt{cls}_1}, \underline{\mathtt{cls}_2} \rangle \in \underline{Aggregate} \Rightarrow \langle \underline{\mathtt{cls}_1}, \underline{\mathtt{cls}_2} \rangle \in \underline{Member}$$

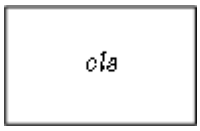
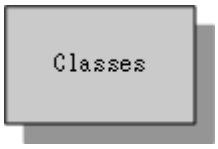
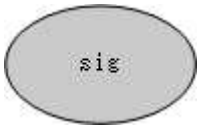
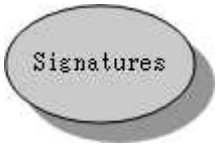
The Axioms of Class-Based Programs require that the design model—the abstract representation of a program—does not violate some inherent principles of object-oriented programming.

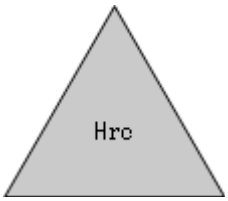
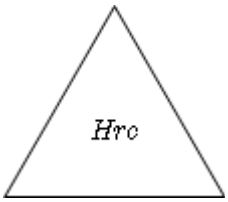
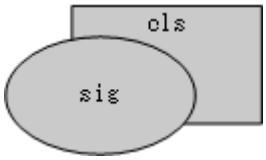
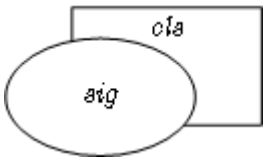
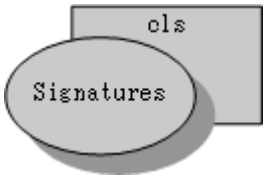
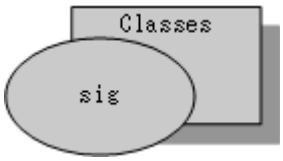
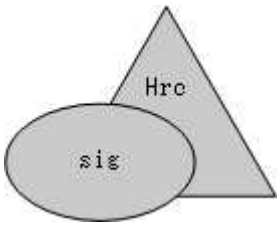
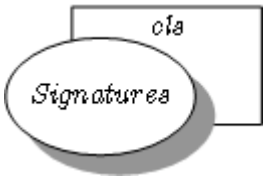
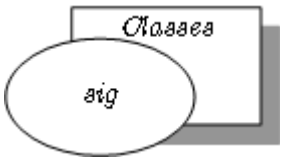
3. LePUS3 and Class-Z

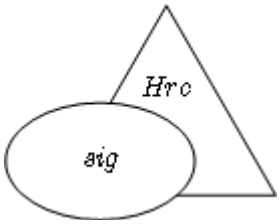
This section is concerned with the actual specification languages. Specifications are spelled out using terms, which stand for entities, and formulas, which impose constraints on entities.

3.1. Terms

Terms stand for entities. Each term is either a constant or a variable term: **Constant terms** represent specific entities where, as a general rule, the constant class/signature/hierarchy X is assigned to the class/signature/hierarchy entity X , and **variable terms** range over entities. Each term has a **type** and **(term) dimension** as specified in the following is a list of symbols for the terms in LePUS3 and Class-Z:

Type	Symbol in LePUS3	Symbol in Class-Z	Symbol name
CLASS		cls	0-dimensional class constant
		cls	0-dimensional class variable
PCLASS		$\mathrm{Classes}$	1-dimensional class constant
		$Classes$	1-dimensional class variable
SIGNATURE		sig	0-dimensional signature constant
		sig	0-dimensional signature variable
PSIGNATURE		$\mathrm{Signatures}$	1-dimensional signature constant
		$Signatures$	1-dimensional signature variable

HIERARCHY		Hrc	1-dimensional hierarchy constant
		*Hrc*	1-dimensional hierarchy variable
METHOD		sig⊗cls	0-dimensional method constant term
		*sig*⊗*cls*	0-dimensional method variable terms
PMETHOD		Signatures⊗cls	1-dimensional superimposition (method) constant terms
		sig⊗Classes	
		sig⊗Hrc	
		*Signatures*⊗*cls*	1-dimensional superimposition (method) variable terms
		*sig*⊗*Classes*	

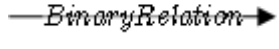
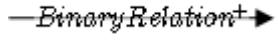
		$\mathtt{sig} \otimes Hrc$	

The type $\mathbb{HIERARCHY}$ is a subtype of $\mathbb{PCLASS}$; in other words, every term of type $\mathbb{HIERARCHY}$ is also a term of type $\mathbb{PCLASS}$. Note that superimpositions can also mix variables with constants, in which case they yield a method variable of the respective dimension and type.

> For sample terms and their semantics see the examples in §1: Terms in [Nicholson et al. 2007, Part II].

3.2. Relation and predicate symbols

For each relation $\underline{Relation}$ we assign the relation symbol $Relation$. The symbol $Relation^+$ is assigned to the transitive closure of binary relation $\underline{Relation}$. The following relation and predicate symbols are admitted:

Symbol in LePUS3	Symbol in Class-Z	symbol name
	$UnaryRelation$	Unary relation symbols
	$BinaryRelation$	Binary relation symbols
	$BinaryRelation^+$	Transitive binary relation symbol
	ALL	ALL predicate symbol
	$TOTAL$	TOTAL predicate symbol
	$ISOMORPHIC$	ISOMORPHIC predicate symbol

3.3. Formulas

Well-formed formulas, or in short formulas, employ predicate symbols, relation symbols, and terms to specify constraints on the entities and relations represented by these symbols.

Definition XII:

Let $UnaryRelation$ be a unary relation symbol, $BinaryRelation$ a binary (possibly transitive) relation symbol, t_1 and t_2 terms of dimension 0. Then the following are **ground formulas**:

Ground formulas in Class-Z	Ground formulas in LePUS3
$UnaryRelation(t_1)$	A unary relation symbol placed over the term t_1
$BinaryRelation(t_1, t_2)$	A binary relation symbol connecting the term t_1 to the term t_1

Let τ_1 and τ_2 be terms, T_1 and T_2 terms of dimension 1. Then the following are called **predicate formulas**:

Predicate formulas in Class-Z	Predicate formulas in LePUS3
$ALL(UnaryRelation, T_1)$	An ALL predicate symbol marked with $UnaryRelation$ placed over T_1
$TOTAL(BinaryRelation, \tau_1, \tau_2)$	A TOTAL predicate symbol marked with $BinaryRelation$ connecting τ_1 to τ_2
$ISOMORPHIC(BinaryRelation, T_1, T_2)$	An ISOMORPHIC predicate symbol marked with $BinaryRelation$ connecting T_1 to T_2

Note that in LePUS3 we do not distinguish between the ground formula $BinaryRelation(t_1, t_2)$ and the predicate formula $TOTAL(BinaryRelation, t_1, t_2)$. Since both formulas are satisfied under the same conditions, there is no ambiguity here.

For sample Class-Z formulas and their semantics see the examples in §2: Ground Formulas and §3: Predicate Formulas in [Nicholson et al. 2007, Part II].

3.4. Specifications

A specification is either a Codechart, a statement in LePUS3, or a Schema, a statement in Class-Z.

Definition XIII:
A **Codechart** consists of a set of terms and well-formed formulas in LePUS3.

You can find many sample Codecharts in lepus.org.uk/spec/ and in lepus.org.uk/ref/legend.

Definition XIV:
A **Class-Z Schema** is an expression of the following form:

$SchemaName$

$declaration$: TYPE

$declaration$: TYPE

...

$formula$

$formula$

...

where each

- $declaration$ is a comma-separated list of constants and variables
- TYPE is a type symbol
- $formula$ is a well-formed formula in Class-Z

You can find many sample schemas in lepus.org.uk/spec and in lepus.org.uk/ref/legend.

We also distinguish between two kinds of specifications: those that contain variables and those that do not.

Definition XV:
A specification which contains no variables is called a **closed specification**, otherwise it is called an **open specification**.

For example, a program is only modelled using closed specifications consisting of 1- and 0-dimensional class, hierarchy, and signature constants.

For example, the specifications of any design pattern, in particular the specifications of all the 'Gang of Four' design patterns are open. (See for example the "'Gang of Four' Companion" [Eden et al. 2007].)

4. Truth conditions

Truth conditions describe the circumstances in which a specification Ψ is satisfied by a design model $\mathfrak{M}$ (also '$\mathfrak{M}$ models Ψ'). We follow the standard Tarski truth condition style, which if satisfied we write

$$\mathfrak{M} \models \Psi$$

The satisfaction of a specification is first defined for specifications that do not contain variables (closed specifications.) The satisfaction of specifications with variables (open specifications) is defined based on that.

The following notational conventions are used in the definitions below: $UnaryRelation$ is a unary relation symbol; $BinaryRelation$ is a binary relation (possibly transitive) symbol; t , t_1 and t_2 are 0-dimensional constant terms; T , T_1 and T_2 are 1-dimensional constant terms.

4.1. Satisfying closed specifications

An closed specification Ψ is satisfied by a design model $\mathfrak{M}$, written $\mathfrak{M} \models \Psi$, iff all the terms in Ψ have an interpretation in $\mathfrak{M}$ and all the formulas in Ψ are satisfied by $\mathfrak{M}$. The specific conditions depend on the formulas as laid as follows. Detailed verification examples and counter-examples are given the document: "Sample Models" [Nicholson et al. 2007].

Definition XVI:
A **ground formula** is satisfied by design model $\mathfrak{M} = \langle \mathbb{U}_*, \mathbb{R}, \mathcal{I} \rangle$ under the following conditions:

- $\mathfrak{M} \models UnaryRelation(t)$ iff $\mathcal{I}(t) \in \underline{UnaryRelation}$
- $\mathfrak{M} \models BinaryRelation(t_1, t_2)$ if one of the following conditions hold:
  - $\langle \mathcal{I}(t_1), \mathcal{I}(t_2) \rangle \in \underline{BinaryRelation}$, or
  - **Subtyping**: There exists some class of dimension 0 $\underline{\mathtt{subcls}}$ in $\mathbb{U}_*$ such that
    $\langle \mathcal{I}(t_1), \underline{\mathtt{subcls}} \rangle \in \underline{BinaryRelation}$ and $\langle \underline{\mathtt{subcls}}, \mathcal{I}(t_2) \rangle \in \underline{Inherit}^+$
    or
    there exists some class of dimension 0 $\underline{\mathtt{sprcls}}$ in $\mathbb{U}_*$ such that
    $\langle \underline{\mathtt{sprcls}}, \mathcal{I}(t_2) \rangle \in \underline{BinaryRelation}$ and $\langle \mathcal{I}(t_1), \underline{\mathtt{sprcls}} \rangle \in \underline{Inherit}^+$

For example, the formula $Member(\mathrm{a},\mathrm{b})$ is satisfied not only when a member of class b is defined inside class a but also by a program which defines a member of class c which inherits from class b inside class a.

For example, the ground formula $Inherit^+(\mathrm{a},\mathrm{b})$ is semantically equivalent to the ground formula $Inherit(\mathrm{a},\mathrm{b})$.

For sample Class-Z ground formulas and their semantics see the examples in §2: Ground Formulas in [Nicholson et al. 2007, Part II].

For subtyping, see Example 7:B in [Nicholson et al. 2007, Part I].

Predicates offer us means for imposing constraints on the relations that may exist between entities. The conditions for satisfying each predicate formula are defined below using the conditions for satisfying ground formulas.

The truth conditions for predicate formulas are defined below for 1-dimensional term arguments. But 0-dimensional term arguments for predicate formulas are also allowed, where each 0-dimensional term argument t is treated as representing the singleton set $\{ \mathcal{I}(t) \}$. For example:

$$\mathfrak{M} \models TOTAL(BinaryRelation, t, T)$$

iff

$$\mathfrak{M} \models TOTAL(BinaryRelation, \{ t \}, T)$$

where $\mathcal{I}(\{t\}) = \{\mathcal{I}(t)\}$.

Definition XVII:
An **ALL** predicate formula of the form $ALL(UnaryRelation, T)$ is satisfied by design model $\mathfrak{M} = \langle \mathbb{U}_*, \mathbb{R}, \mathcal{I} \rangle$ iff for each $\underline{e}$ entity in $\mathcal{I}(T)$: $\mathfrak{M} \models UnaryRelation(e)$.

For example, the predicate formula $ALL(Abstract, \mathtt{Operations} \otimes \mathtt{collection})$ requires that all the methods in class `Collection` with a signature represented by the elements of $\mathtt{Operations}$ are abstract.

See also Example 15, Example 16, and Example 17 in [Nicholson et al. 2007, Part II].

Definition XVIII:
A TOTAL predicate indicates the existence of a total functional relation from the concrete elements of one set to another. More formally, a **TOTAL** predicate formula of the form $TOTAL(BinaryRelation, T_1, T_2)$ is satisfied by design model $\mathfrak{M} = \langle \mathbb{U}_*, \mathbb{R}, \mathcal{I} \rangle$ iff $\mathcal{I}(T_1)$ does not consist solely of abstract methods, and for each $\underline{e}_1$ entity in $\mathcal{I}(T_1)$ that is not an abstract method there exists some $\underline{e}_2$ entity in $\mathcal{I}(T_2)$ such that $\mathfrak{M} \models BinaryRelation(e_1, e_2)$.

The formula $TOTAL(BinaryRelation, T_1, T_2)$ thus indicates that the relation $BinaryRelation$ is a total functional relation from the set of entities represented by the first term T_1 (with the exception of abstract methods) to the set of entities represented by the second term T_2.

For example, the formula $TOTAL(Member, \{cls_1, cls_2\}, \mathtt{object})$ requires that class `cls1` and class `cls2` have each a member of type `Object` (or of some class that inherits from it.)

See also Example 18, Example 19, and Example 20 in [Nicholson et al. 2007, Part II].

Definition XIX:
An ISOMORPHIC predicate indicates the existence of a bijective (1:1 and onto) functional relation between the concrete elements of two sets. More formally, an **ISOMORPHIC** predicate formula of the form $ISOMORPHIC(BinaryRelation, T_1, T_2)$ is satisfied by design model $\mathfrak{M} = \langle \mathbb{U}_*, \mathbb{R}, \mathcal{I} \rangle$ iff there exists a pair $\langle \underline{e}_1, \underline{e}_2 \rangle$ where $\underline{e}_1 \in Concrete(\mathcal{I}(T_1))$ and $\underline{e}_2 \in Concrete(\mathcal{I}(T_2))$ such that:

- $\mathfrak{M} \models BinaryRelation(e_1, e_2)$; and
- $\mathfrak{M} \models ISOMORPHIC(BinaryRelation, T_1\text{-}e_1, T_2\text{-}e_2)$ unless both $T_1\text{-}e_1$ and $T_2\text{-}e_2$ are empty.

where $\mathcal{I}(T\text{-}e) = \mathcal{I}(T) - \mathcal{I}(e)$ and $Concrete(S)$ stands for the subset of set S which consists of all and only non-abstract entities in S.

The formula $ISOMORPHIC(BinaryRelation, T_1, T_2)$ thus indicates that there exists a subset of the relation $BinaryRelation$ which is a bijective functional (one-to-one and onto) relation from the set of

concrete entities represented by the term T_1 to the set of entities represented by the term T_2.

For example, the formula $ISOMORPHIC(Member, \{a_1, a_2\}, \{b_1, b_2\})$, if all classes are non-abstract, requires that either that class `A1` has a member of class `B1` (or of its subtypes) and class `A2` has a member of class `B2` (or of its subtypes), or that class `A1` has a member of class `B2` (or of its subtypes) and class `A2` has a member of class `B1` (or of its subtypes).

See also Example 21, Example 22, Example 23, Example 24, and Example 25 in [Nicholson et al. 2007, Part II].

4.2. Satisfying open specifications

The truth conditions for open specifications require the notion of an assignment.

Definition XX:
An **assignment** g from variables $v_1 \ldots v_n$ to constants $c_1 \ldots c_k$ is a function
$g : \{v_1, \ldots, v_n\} \rightarrow \{c_1, \ldots, c_k\}$

A well-formed assignment associates each variable with a constant of same type and dimension. Assignments are used so as to associate variables in generic specifications, such as design patterns, to constants representing specific elements of concrete programs.

For example, an assignment $JavaIterator$ from the Iterator design pattern [Eden et al. 2007] to the abstract semantics of `java.util` may read:

$$JavaIterator(Aggregates) = \{\texttt{collection}, \texttt{linkedList}, \texttt{hashSet}\}$$
$$JavaIterator(Iterators) = \{\texttt{iterator}, \texttt{linkedList.ListItr}, \texttt{hashMap.KeyIterator}\}$$
$$JavaIterator(next) = \texttt{next}$$
$$JavaIterator(newItr) = \texttt{iterator()}$$
$$JavaIterator(element) = \texttt{object}$$

We say that the open specification $\Phi[v_1, \ldots, v_n]$, where $v_1 \ldots v_n$ are all the (distinct) variables in Φ , is satisfied by a design model $\mathfrak{M} = \langle \mathbb{U}, \mathbb{R}, \mathcal{I} \rangle$ under the assignment
$g : \{v_1, \ldots, v_n\} \rightarrow \{c_1, \ldots, c_k\}$, written

$$\mathfrak{M} \models_g \Phi[v_1, \ldots, v_n]$$

iff

$$\mathfrak{M} \models \Phi[g(v_1)/v_1, \ldots, g(v_n)/v_n]$$

where $\Phi[\, g(v_1)/v_1, \ldots, g(v_n)/v_n \,]$ is that closed specification which results from the consistent replacement of variable v_1 with the constant $g(v_1)$, ... and the consistent replacement of variable v_n with the constant $g(v_n)$ in the open specification Φ .

> For example, let $IteratorPattern[\, Aggregates, Iterators, next, newItr, element]$ be the specification of the the Iterator design pattern [Eden et al. 2007], design model $\mathfrak{JavaUtil}$ is the abstract semantics of the `java.util` package, and $JavaIterator$ be the assignment from $\{\, Aggregates, Iterators, next, newItr, element\}$ to the universe of $\mathfrak{JavaUtil}$. Then we write
>
> $$\mathfrak{JavaUtil} \models_{\mathsf{JavaIterator}} IteratorPattern$$
>
> to indicate that `java.util` satisfies (also 'models' or 'implements') the Iterator design pattern.

5. Consequences

Proposition 1: LePUS3 and Class-Z are proper subsets of FOPL.

Proposition 2: $\underline{Inherit}^+$, the transitive closure of the $\underline{Inherit}$ relation, is a strict order on $\underline{Class}$

Proposition 3: $\underline{SingatureOf}$ is a functional relation.

6. Acknowledgements

Many thanks go to Ray Turner for his continuous help.

References

- Amnon H. Eden. "Formal Specification of Object-Oriented Design." *Proc. Int'l Conf. Multidisciplinary Design in Engineering CSME-MDE 2001* (21–22 Nov. 2001), Montreal, Canada. [.pdf]
- Amnon H. Eden. *Object-Oriented Modelling*. Under preparation.
- Amnon H. Eden, Jonathan Nicholson, Epameinondas Gasparis. "The LePUS3 and Class-Z companion to the 'Gang of Four' design patterns." Technical report CSM-472, ISSN 1744-8050 (2007), Department of Computer Science, University of Essex. [.pdf]
- Erich Gamma, Richard Helm, Ralph Johnson, John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Reading: Addison-Wesley, 1995.
- Michael R.A. Huth, Mark Ryan. *Logic in Computer Science*. Cambridge: Cambridge University Press, 2000.

- Jonathan Nicholson, Amnon H Eden, Epameinondas Gasparis. "Verification of LePUS3/Class-Z Specifications: Sample models and Abstract Semantics for Java 1.4 (Part I; Part II)." Department of Computer Science, University of Essex, Tech. Rep. CSM-471, ISSN 1744-8050 (2007). [.pdf]
- J. Michael Spivey. *The Z Notation: A Reference Manual*. Hertfordshire: Prentice-Hall, 1992.