SOFTWARE ENGINEERING – DESIGN I

3. Schemas and Theories

The aim of this course is to learn how to write formal specifications of computer systems, using classical logic. The key descriptional technique is that of a *logical theory:* you write a specification as a logical theory, and then the final system implementation is seen as giving a model of that theory.

However, the specification is not entirely *pres*criptional – "This is how things shall be". The implementation will have to work in the real world and so part of the specification will be a logical *de*scription – "This is how things are" – of those aspects of the real world that are relevant to the specified system. In this respect specification is like a scientific theory. A good specification will include sharp non-obvious assumptions about how the real world actually behaves, and so will lay itself open to being proved wrong in the light of experience or closer examination. A specification that is not potentially falsifiable in this way is almost certainly too vague to be useful.

This course and **Z**

One widely used language for specifying computer systems is **Z**, and much of the notation used in this course is borrowed from **Z**. However, I must stress that it is not the purpose of this part of this course to teach you how to use **Z** but rather to show you how to use logical ideas in specifications, and in particular the idea of a logical theory. I shall wherever possible use the notation that most of you are already familiar with, just borrowing a few key good notational ideas from **Z**.

Schema notation

We borrow the schema notation of **Z** to present "many-sorted first-order theories". This is not exactly the way **Z** itself thinks of schemas, but it is not far off. Here is what a schema looks like:

Sorts
↓

Theory name [X, Y, …]	
$f: X \times Y \to Z$ $P \subseteq U \times V$	*Signature* (function and predicate symbols)
ϕ ψ	*Axioms* (called the *predicate* in **Z**)

In the above example, "f" is a function that assigns to each pair of the Cartesian product X×Y elements of the sort Z. The symbol "P" is instead a relation, i.e. a set of pairs from the Cartesian product U×V. The symbols ϕ and ψ denote instead logical formulae (or axioms), written using the symbols specified in the signature (i.e. f and P). In the specification language **Z**, schemas are used to specify systems' states. The part above the line, called *declaration*, describes the systems' variables (i.e. variables, functions, predicates or relations of classical logic), whereas the part below the line, called *predicate*, specifies constraints on the variables' values (e.g. formulae or axioms about the predicates or about the values that the functions can assume).
We still have to explain what many-sorted first order theories are.

Logical Theories

A theory comprises a vocabulary of symbols (such as functions and predicates) together with some axioms; and a model of the theory is a structure that interprets the symbols in the vocabulary in such a way as to make the axioms hold. We are now going to extend these concepts in some ways, so we shall give the definitions explicitly.

A *many-sorted vocabulary* or *signature* comprises –

- a set of *sorts*
- a set of *function symbols,* each with a given *function arity*
- a set of *predicate symbols,* each with a given *predicate arity*

For those of you who are not familiar with this terminology, a predicate symbol with arity *n* is essentially a relation with arity *n*, namely a set of tuples of n elements. In the example above P is a predicate symbol with arity 2, which can also be seen as a binary relation. It denotes a set of pairs (u,v) where the first element "u" of the pair is an element from the sort U and the second element "v" of the pair is an element from the sort V. Notice that the ordering of the elements in the pair counts. So the pair (u,v) is different from the pair (v,u).

"Many-sorted" refers to the possibility of having numerous sorts. The sorts work like types in a computer programming language, so that some expressions and formulae can be rejected as meaningless simply because they are not "well-typed". To make this work, each function or predicate symbol has an *arity* – a declaration of the sorts of its arguments and (for a function) result.

Predicate arities: The arity of a predicate P is a finite list of sorts, $Xs = [X_1, \ldots, X_n]$ (say). Then P can be used only with n arguments, $P(x_1, \ldots, x_n)$, where each expression x_i has sort X_i. To indicate that P has this arity, we write $P \subseteq X_1 \times \ldots \times X_n$.
Function arities: The arity of a function f is a pair (Xs, Y) where Xs is a list of sorts and Y is a sort. Then f can only be used with n arguments, $f(x_1, \ldots, x_n)$, and its result has sort Y. To indicate that f has this arity, we write $f: X_1 \times \ldots \times X_n \to Y$.
Terms and formulae: Once we have a signature or vocabulary, we can build up terms and logical formulae as illustrated in the lecture. There are two novelties between first-order logic and many-sorted logic. First, in a function or predicate, the arguments have to have the sorts specified by the arities; and, second, variables must have their sorts specified. For instance, for a quantified variable x we write $\forall x{:}X. \ldots$ or $\exists x{:}X. \ldots$ to specify that its sort is X.

Structures: Suppose we are given a signature. Then a *structure* for it comprises –

- for each sort X, a corresponding set $[\![X]\!]$, the *carrier* or *domain* for X.
- for each predicate symbol $P \subseteq X_1 \times \ldots \times X_n$, a corresponding subset $[\![P]\!]$ of the Cartesian product
  $[\![X_1]\!] \times \ldots \times [\![X_n]\!]$.
- for each function symbol $f: X_1 \times \ldots \times X_n \to Y$, a corresponding function $[\![f]\!]$ from the Cartesian product $[\![X_1]\!] \times \ldots \times [\![X_n]\!]$ to $[\![Y]\!]$.

(We shall never really use these "semantic brackets" $[\![$ and $]\!]$. Just for this definition, we want to make a clear distinction between the syntactic symbols X, f, P, etc. and their set-theoretic meanings $[\![X]\!]$, $[\![f]\!]$,

⟦P⟧, etc. However, in practice we introduce the syntax just so that we can use it as notation for the sets, and the semantic brackets are a waste of ink.)

Once these basic ingredients of a structure are given, every term and formula gains a meaning within the structure. A term of sort Y, with free variables $x_1, \ldots, x_n$ of sorts $X_1, \ldots, X_n$, gains meaning as a function from $⟦X_1⟧ \times \ldots \times ⟦X_n⟧$ to ⟦Y⟧, and a formula with those free variables, gains meaning as a subset of $⟦X_1⟧ \times \ldots \times ⟦X_n⟧$. For instance, if x and y have sorts X and Y, then

⟦∀x. P(x,y)⟧ = {b ∈ ⟦Y⟧: for every a ∈ ⟦X⟧, (a,b) ∈ ⟦P⟧}

For those who want to know more: Nullary predicates or functions are those predicate and functions these have no arguments – n = 0 in the arity. The corresponding Cartesian product of no sets is **1**, a set with just one element ( ). (How many tuples are there with no components? Just one.) **1** has two subsets, the empty set **Ø** and the set itself {( )}, which are #possible values for ⟦P⟧ when P is nullary. They can be taken as **false** and **true** respectively. A nullary function has no arguments but does have a result. It is the same as a constant.

Theories: Given a signature, a *theory* is a set of logical sentences (formulae without free variables) constructed from it. These logical sentences are called the *axioms* of the theory.

Schemas: The sorts, symbols and axioms of a theory can be written down in the form of a schema as illustrated earlier. The schema also has a place for a *name* for the theory, if desired.

Models: Suppose we are given a theory, and a structure for its vocabulary. Then that structure is a *model* of the theory if every sentence in the theory is interpreted as **true**. More pedantically, if ϕ is any sentence in the theory, then ⟦ϕ⟧ is **true**.

Questions

1. How many sorts do you need for first-order logic?
2. What is a theory with no sorts at all?

Special Purpose Sets as Carriers or Domain of Discourse.

New sorts from old

Sometimes symbols have a particular intended mathematical meaning, for instance –

ℕ	natural numbers {0,1,2,3,…}
ℤ	integers (all whole numbers including negative ones)
ℚ	rational numbers
bool	{**false**, **true**}
1 {( )}	(set whose only element is the unique empty tuple)

Sometimes also, sorts are intended to be constructed from others in a particular way. We shall indicate this using the obvious set-theoretic notation. For instance,

X×Y	Cartesian product of X and Y
X+Y	disjoint union of X and Y
seq X	finite sequences of elements of X
𝔽X	finite power set of X (i.e. the set of all finite subsets of X)

When a vocabulary includes constructed sorts like this, a structure for it must have its carriers constructed in the corresponding way. The same goes for function and predicate symbols that have a standard mathematical meaning, such as

+: ℕ×ℕ →ℕ
cons: X ×seq X →seq X
∪: **𝔽X** × **𝔽X** → **𝔽X**.

Examples

A theory is really just a formal way of describing what the models are: so in the following examples we describe the models first informally and then formally, by schemas.

(i) A model is a non-negative even integer:

_____ Even _____
c: ℕ
∃a: ℕ. c = 2*a

Note that we don't name the constructed sort ℕ after the schema name, as Even [ℕ]. That place is reserved for the **"primitive" sorts** that are not constructed out of others and for not special purpose sorts.

(ii) A model is an infinite stream of characters which, when divided into lines by a given "newline" character, has all lines shorter than a given maximum length:

_____ Lines_____
linemax: ℕ
NL: **char** s: ℕ → **char**
∀m: ℕ. ∃n: ℕ. (m < n ≤ m+linemax ∧ s(n) = NL)

In the schema Lines, NL represents the "newline" character, m is the length of a line excluded the NL character, and n is the position in the line of the newline character.

(iii) A model is a set from the sort people, and two unary functions, pa and ma, such that no element can be both a pa of some other element and a ma of some other element.

_____ People [people]_____
alive ⊆ people
pa: people → people
ma: people → people
∀x, y: people. (pa(x) ≠ ma(y))

Note that although the naming suggests that the carrier of "people" should be the set of people, this is not something that can be imposed mathematically: "people" is not like "ℕ". The real-world characteristics of the set of people are not mathematically defined. Part of the art of specification is to

formalize – in the schema – those characteristics that are going to be important when considering the computer system we are specifying. The characteristics I've given here – the alive predicate and the pa and ma functions with the further axiom – are unlikely to be exactly the relevant characteristics in most applications, but they serve as examples.

When we find a model of People, in practice the carrier of people is not intended to be a mathematically defined set but an informal "real-world" set. Of course, we don't expect to have a list of all possible people in the world. What we do expect from one of these informal sets is an understanding of

(i) how to know when we've found an element, and
(ii) how to know when two elements we've found are equal or (usually) unequal.

Without at least these, we cannot make much sense of the formal logic.

More on schemas

Schema inclusion: We can "include" one schema in the signature of another just by writing its name. For instance,

Emp

People
employees: $\mathbb{F}$people

is equivalent to

Emp [people]

alive $\subseteq$ people
pa: people $\rightarrow$ people
ma: people $\rightarrow$ people
employees: $\mathbb{F}$people
$\forall$x,y: people. (pa(x) $\neq$ ma(y))

Note that we have specified employees as a *finite* subset of people. This has the important consequence that we are able to get a complete list of all the employees, and record them in a database if necessary, unlike the case with people itself. We shall be very careful to specify finiteness where appropriate.

Schema conjunction: Let S and T be two schemas. Their *conjunction* S$\wedge$T is constructed by putting all the vocabulary and axioms of S and T together:

S$\wedge$T

S	(two schema inclusions)
T	

What is a model of S$\wedge$T? If the vocabularies of S and T are disjoint, then a model of S$\wedge$T is a pair (M,N) where M and N are models of S and T. It is more complicated if S and T have symbols in common, but this is out of the scope of this course.

Example

Suppose we have Emp as before, and

Cust

People
customers: Fpeople

Then a model for Emp$\wedge$Cust is a single set people (with pa and ma operations) together with two finite subsets, employees and customers.

Change and non-change: Special examples of schema conjunction can occur for – in effect – conjoining a schema S with itself. To see this, we first need the idea of schema decoration. When we conjoin a schema S with itself, the symbols, which are defined in the same way in both versions of S, need to be distinguished. "Decorating" means represent all symbols of a "second instance" of S with some mark to distinguish them from the original symbols of a "first instance" of S. (We shall assume that S does not already contain that mark in any way that might cause confusion.) For instance, with $_0$ as the decorating mark, People$_0$ is

People$_0$ [people$_0$]

pa$_0$: people$_0$ $\rightarrow$ people$_0$
ma$_0$: people$_0$ $\rightarrow$ people$_0$
$\forall$x,y:people$_0$. pa$_0$(x) $\neq$ ma$_0$(y)

For any schema S, we write

ΔS

S
S'

or ΔS = S$\wedge$S'. Then a model of ΔS is simply two models of S.

Operations

The Δ notation is very useful when describing *change* (and in fact that is why the symbol Δ was chosen): if the possible states of a system are the models of a schema S, then a change in the system, described as the states before and after, is a model of ΔS. We shall adopt the usual **Z** convention that unprimed variables refer to the state before, primed ones to the state after.

Example

Suppose a computer program uses two variables, x and y, with natural number values. A state is a model of

State

x,y: $\mathbb{N}$

Then the Swap operation can be described by

_____ Swap _____

ΔState
x'=y y'=x

Some common **Z** conventions to specify operations with schemas.

- state variables are written unprimed and primed for before and after
- input variables (or arguments) are decorated with "?"
- output values (or results) are decorated with "!"

For example,

_____ State _____

eps: float
eps > 0

_____ Sqrt _____

ΞState x?, r!: float
pre: x? ≥ 0 post: (|r!^{2} – x?| < eps) ∧ (r! ≥ 0)

Note that the primes, the decorations "?" and "!", and the annotations "pre:" and "post:" mean nothing special if you think of a schema as a presentation of a theory. On the other hand, informally they do suggest other relevant theories. For instance, the state before, the inputs and the preconditions together specify a domain of definition for the operation:

_____ Sqrt_before _____

State x?: float
x? ≥ 0

No change

One of the commonest ways of specifying how things change is to say that they don't! This can be expressed using the Ξ notation (Ξ, a capital Greek letter "xi", is chosen to look like a big equivalence sign, ≡).

_____ ΞS _____

ΔS
equalities between all the symbols of S and their corresponding primed versions

On its own, this is little use: for a model of ΞS is exactly the same as a model of S (why?). However, it is useful when included in other schemas.

Examples

Suppose we want to specify an update operation on Emp when a new employee is taken on. The set of real world people isn't affected.

_____ AddEmp _____

ΔEmp ΞPeople new_emp?: people
employees' = employees ∪ {new_emp?}

Another example: suppose Lift is a schema whose models are lifts. (This isn't quite right. Models are mathematical structures. But any real-world object worthy of the name "lift" will have various essential properties that can be abstracted as mathematical structure. ***"Abstraction" is to ignore inessential properties such as the color of the ceiling or the typefont on the manufacturer's nameplate***.) Suppose also the schema Lift includes a schema CallButtons. If we have two lifts side by side serving the same floors then they ought to share their call buttons, and this can be specified by

_____ TwoLifts _____

ΔLift ΞCallButtons

(This example is very silly! The Δ-Ξ notation is designed for and well-adapted to change over time. In this example it is being used for replication over space, so (1) it's slightly tricky to describe more than two lifts together, and (2) when you come to describe how lifts *do* change over time – and if they don't there's no point in having them – then you can't use Δ and Ξ for it.)

A model for TwoLifts is two lifts, L and L', that share CallButtons.

Exercises

1. In the context of Emp, specify an operation RemEmp to remove an employee. Include as precondition that the employee to be removed is actually an employee.

_____ RemEmp _____

ΔEmp ΞPeople old_emp?: people
pre: old_emp? ∈ employees post: old_emp? ∉employees' employees = employees' ∪ {old_emp?}

Note that the precondition is actually a logical consequence of the postcondition, so as a theory presentation this schema is equivalent to one with the precondition omitted. But informally it carries the suggestion that nothing is guaranteed if you try to perform the operation with the precondition failing.

Schema disjunction

This is a construction on schemas that is very useful for expressing alternatives or exceptions. It is slightly more complicated that schema conjunction.

Suppose schema S includes (or can be considered to include) a schema S' (*not* a decorated version of S, just another schema) such that the only difference between them is that S contains some extra axioms. Suppose similarly T is T' with some extra axioms, and suppose also that S' and T' have compatible vocabularies – any symbols that they share are declared with the same sort in both S' and T' – so that S'∧T' is meaningful. Then the schema disjunction S∨T is

_____ S∨T _____

S'∧T'
(extra axioms of S) ∨ (extra axioms of T)

A model is a model (M,N) of S'∨ T' such that either M is a model of S or N is a model of T.

The easiest case of this is when S' and T' are just the vocabularies of S and T: then S∨T has the merge of the vocabularies and the disjunction of the axioms. Notice that the notation S∨T has avoided mentioning S' and T': in practice you use the notation when it is more or less obvious what S' and T' are expected to be. (**Z** gives rules for "normalizing" arbitrary schemas S, that in effect extract canonical schemas S'. However, the process depends critically on **Z**'s notion of "types", and it is not possible to explain it just in terms of logical theories.)

As an example, consider AddEmp. If we try to add a new employee who's actually already on the payroll, the operation will succeed but it would probably be better if it returned some kind of error or exception message. This can be done with a result variable r! as follows. First, we need to specify a set of possible reports, which we do by

REPORT := {success, already_in}

This is in effect extending our background mathematics by a special purpose sort constructor. Of course, we may want to add in extra possible reports later, as extra elements of REPORT, but that won't be a problem. Now we can use these report elements:

_____ AddEmpSuccess_____

ΔEmp	
ΞPeople	
new_emp?: people	
r!: REPORT	
pre:	new_emp? ∉ employees
post:	employees' = employees ∪ {new_emp?}
	r! = success

_____ AddEmpAlreadyEmployed _____

ΞEmp	
new_emp?: people	
r!: REPORT	
pre:	new_emp? ∈ employees
post:	r! = already_in

AddEmp = AddEmpSuccess ∨ AddEmpAlreadyEmployed

Expressed as one big schema, we should have

_____ AddEmp _____

ΔEmp
ΞPeople
new_emp?: people
r!: REPORT
(new_emp? ∉employees ∧ employees' = employees ∪ {new_emp?} ∧ r! = success)
∨ (new_emp? ∈ employees ∧ r! = already_in ∧ *employees*' = *employees*)

The advantage of this style is that the main part of the operation – the successful case – can be written in a single schema without the distraction of the exceptional cases. These are added on separately as we need for them.

Notice –

- ΞEmp in AddEmpAlreadyEmployed: in the error case, we do not want any change of state. In the combined schema for AddEmp this has to appear as employees' = employees in the predicate. Remember from the general discussion of schema disjunction that each schema has to be considered to be a smaller schema with some extra axioms; in this case the ΞEmp in AddEmpAlreadyEmployed is being considered as ΔEmp with extra axiom employees' = employees.

- The new AddEmp is a defensive specification and so does not need preconditions.