Formal Specifications

- **Course**: *Software Engineering - Design I*
- **Lecturer**: Dr. Alessandra Russo
  - email: ar3@doc.ic.ac.uk
  - office hours: available in my office (Room 560) between 1:30-3:30pm on Wednesday.
- **Duration**: 6 lectures and 3 tutorials

Course: Software Engineering – Design I

Aims and Objectives

This part of the course is about formal specification and its role in software development.

It aims to illustrate:

- what are the basic logical ideas underlying formal specification,
- what are the main components of a formal specification,
- how to write simple formal specifications from some given informal requirements.

Unit 1: Theories and Schemas Slide Number 2

This part of the course is about formal specification and its role in software development. Formal specifications can help make the development of (large) computer systems successful, by helping understand **what** the system is supposed to achieve and **why**.

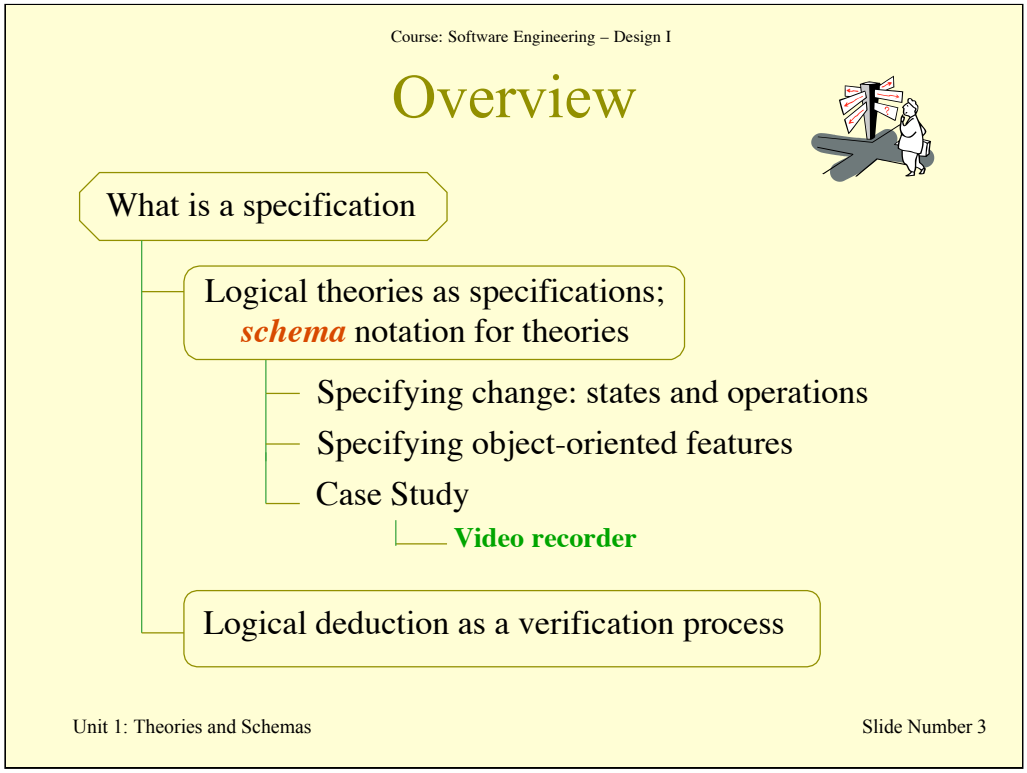
The **aims** are:

- To understand the role that logical theories play in specifying computer systems. Conventional logic will be used, adopting notation similar to that used in the Z specification language, in order to illustrate the basic logical ideas underlying a formal specification.
- To understand what the main components of a formal specification are and why.
- To learn how to write formal specifications.
- To illustrate some of the issues involved in developing formal specification from informal requirements, by looking at an example case study.
- To illustrate how formal specifications and formal derivations can be used to check if a product meets its requirements.

Objectives:

At the end of this part of the course, you should be able to:

- discuss advantages and disadvantages of formal specifications in software development,
- understand the logical ideas underlying a formal specification,
- define logical theories as formal specifications (i.e. using the schema notation) of computer systems, and
- use them in order to verify if the resulting system meets its requirements.



A very brief introduction to what formal specifications are, why and when they are useful for the development of (large) computer systems is given in the additional notes distributed at this lecture.

We will begin with a brief description of a real example of a computer-based system failure and explanation of why formal specifications are important in software development. We'll then concentrate for the next two lectures on summarising basic concepts of logic and logical theories that are going to be needed in this part of the course. For both Computing and JMC students this introduction will be a simple recap of concepts already learned during the first year course on logic.

We will then show how to write **logical theories** as formal specifications. Much of the formal specification notation is borrowed from the Z specification language, for instance the **schema notation**. Please note that it is not the purpose of this course to teach how to use Z, but rather to show some basic features of formal specifications and how they can be expressed using classical logic.

Once we have introduced the concepts of logical theories and schemas, we will see how these concepts can be used to specify

- **change** in a software system (i.e. operations that take a system from one state to another), and
- **object-oriented features** of systems, such as the notions of classes and objects as instances of classes. We'll then look at one example case study in detail to illustrate the entire process of pecifying a (software) system starting from an initial diagrammatical description of the system behaviour.

Reading Material

- Books recommended are:
  - "**The Way of Z**", J. Jacky, Cambridge University Press, 1997.
  - "**Formal Specification Using Z**", L. Bottaci & J. Jones, International Thomson Publishing, 1995.
  - "**Z in Practice**", by R. Barden, S. Stepney & D. Cooper, Prentice Hall, 1994.
  - "**An Introduction to Formal Specification and Z**", B. Potter, Prentice Hall, 1991.
  - "**Safeware: System Safety and Computers**", N. Leveson, Addison-Wesley, 1995.

- Slides and notes, complemented with information given during lectures and tutorials.

The most useful books are those that use the Z language to discuss specification. The first few chapters in each book are basic introductions to formal specification (or more generally to formal methods). But remember that we are not adhering to the strict Z notation.

The first book explains well (re. first three chapters) what a formal specification is, why and and when to use it. The second book is a very simple book. I'm suggesting it mainly because it is an easy introductory reading about mathematical modelling and formal specification. You might also find some of the examples useful to understand the procedure of defining a formal specification of a computer system from given informal user requirements. The third and forth books are general books about specification and Z.

Finally, the last text book is on formal specifications in a more general sense. It provides a collection of interesting real examples of computer-based system failures that could have been avoided if formal specifications had been used during the development cycle. It is listed here for your own general interest.

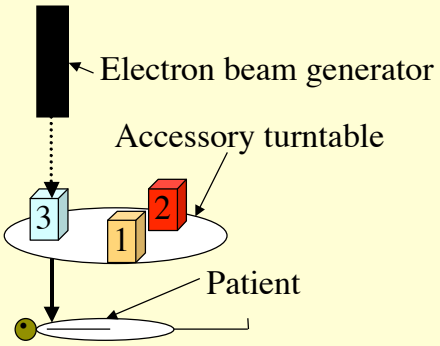
The slides are available on the Web at

http://www.doc.ic.ac.uk/~ar3/lectures/Sed/NewCourseStructure.html,

or following the links from my Web page at http://www.doc.ic.ac.uk/~ar3.

A Computer-initiated Disaster: Therac-25

Radiation therapy machine massively overdosed 6 patients.



Three turntable positions:

1 Low energy electrons

2 High energy electrons

3 "Field light" for positioning purposes.

Software fault:

turntable in wrong position ⇒ unsafe high radiation dose

This case study is about a computer-controlled radiation therapy machine, called Therac-25, which has massively overdosed 6 people. This type of machine consists of both hardware and software components able to accelerate electrons to create high-energy beams that can destroy tumors with a minor impact on the surrounding healthy tissues. In this type of treatment, shallow tissues are treated with low energy accelerated electrons, whereas deeper tissues are treated with electron beams converted into X-ray photons. The special feature of this machine was to be able to deliver photons as well as electrons at various energy levels.

A major hardware component of the system was a turntable, which rotated accessory equipment into the beam path to produce the two therapeutic modes: the electron mode (position 1) and the photon mode (position 2). The third position of the turntable did not involve any beam, but it was used for facilitating correct positioning of the patient under the machine. Therefore proper operation of the machine strictly depended on the positioning of the turntable. Three micro switches monitored the position of the turntable.

In the *electron therapy*, the accelerator beam can be generated at different energy levels. In the photon therapy, only one (high) energy level is available, which is attenuated by a "beam flattener". If the machine produces a photon beam, with the flattener not in position, then the patient will receive an unsafe high output dose. If the turntable is in the wrong position, the beam flattener will not be in place.

When the turntable is in the third position, no accelerator beam should be delivered.

In Therac-25, the computer was responsible for the positioning of the turntable.

Unsafe situations

A high energy electron beam can be needed but mustn't hit patient directly.

High electron beam + turntable in position 1 or 3 **= unsafe**

- Predecessors Therac-6, Therac-20 had hardware interlocks. Therac-25 relied only on software checks.
- **Unsafe situations weren't detected and patients were killed.**
- **Main Cause**:

  Unsafe architecture & lack of specs → software bugs

As mentioned in the previous slide, to deliver X-ray (photon) treatments, a high energy electron beam is generated and flatted by a beam flattener attached to the turntable (i.e. position 2). One of the safety properties of the system is that such a high energy beam should never hit the patient directly. Positions 1 or 3 are then wrong positions of the turntable, when the machine is programmed to deliver an X-ray treatment. **Extremely unsafe circumstances would therefore arise when the system is in X-ray mode, delivering a high energy electron beam and the turntable is in one of these wrong positions.**

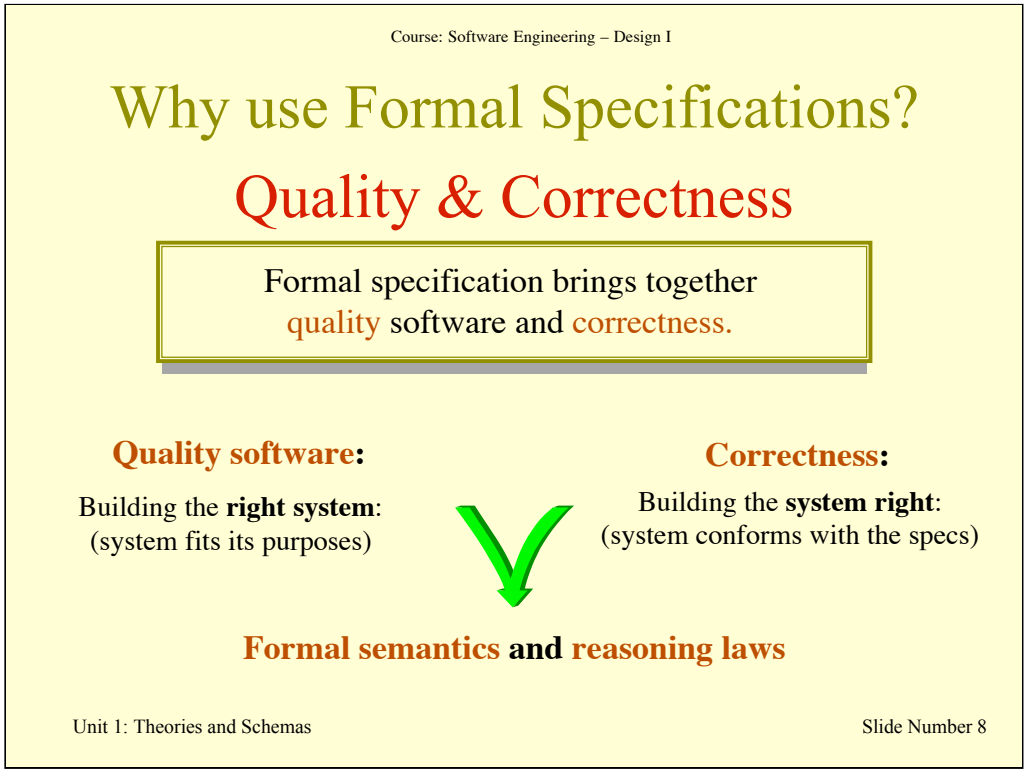
Previous versions of Therac-25, called respectively Therac-6 and Therac-20 used only hardware interlocks to guarantee that the turntable and attached equipment were in the correct position before starting any treatment. The software component merely added convenience (e.g. interface) to the existing hardware system. In Therac-25, hardware interlocks were fully substituted by software checks. The software in Therac-25 had, therefore, more responsibility for maintaining safety than its predecessors. BUT software from the previous version systems were used in Therac-25 and some unsafe situations were not detected. As a consequence six accidents occurred of massive overdose and, for some, this resulted in death of the patient.

The main failing factors were that Therac-25 was built by taking into account design features and modules of Therac-6 and by re-using some of the software routines developed for Therac-20. However, hidden bugs and problems of these two predecessors had been covered in these previous systems by their hardware interlock mechanisms. This wasn't the case for Therac-25.

The unsafe architecture of Therac-25 and the lack of formal specifications of the software developed for Therac-20 hadn't therefore revealed existing software bugs, which were aggravated by problems such as quality control, lack of good documentation of previous failures, etc.

Lesson Learned

- Formal specifications and rigorous analysis of existing software with respect to the new system architecture would have highlighted the problems.
- A disaster would have been avoided.



What we are interested in software engineering is to have *quality* software, i.e. software that fits its purpose and that does something of value for the user, as well as *correct* software, i.e software that respects (satisfies) the specifications.

Formal specification brings these two aspects together. It helps achieve quality by facilitating an understanding of the user requirements and therefore helps to not loose sight of user needs. At the same time, it provides a means for proving that the system satisfies the users' requests in a correct way.

Specifications can be used to both **validate** and **verify** a (computer) system. Its precise semantics and reasoning laws enable these two tasks to be performed. Users' needs can be formalized and proved to be satisfied by the specifications of the system. In the same way, program code can be proved to verify the specifications. In your first year course of reasoning about programs, you have already seen some simple examples of verification, when you were proving that (little) procedures satisfy their pre- and post-conditions. The definition of specifications helps implementers to focus on small parts of the system and forget about the rest, just working blindly to what has been decided. The verification of each part of the system with respect to its specifications contribute towards the correctness of the entire system.

Theories and Schemas

This lecture aims to:

- Recall some concepts from last year: **structures** and **models**.
- Define the notion of logical theories as specifications.
- Define the ***schema*** notation used to represent logical theories.
- Define the notion of ***schema inclusion***.

The remaining of this lecture and the next one aim (1) to represent some basic concepts of classical logic. We will start with recalling some basic concepts from your first year course on classical logic. We will in particular see what a logical theory, structures and models are in standard classical logic. These include the notions of *logical theories*, *structures* and *models*. (2) Then, we will see, via some example, how classical logic theories can be written as formal specifications. Through this examples we will gradually introduce the notion of a **schema notation**, borrowed from Z, and show how schemas can be used to write logical theories as specifications. We will then define and illustrate with an example the fundamental notion of **schema inclusion**. Finally we will briefly see how to extend classical first-order logic in order to use logical theories for specifying real-world systems.

By the end of this Unit 1, you will know most of the formal notation that we are going to use in the remaining four lectures, and will have learned how to define schemas to specify simple system states.

Course: Software Engineering – Design I

Propositional Logic

Language:

- logical operators: ∧, → , ¬, ∨, have a fixed meaning (truth tables)
- extra-logical symbols: P, Q, R, ... propositional letters don't have a fixed meaning

Theory:

A set of logical formulae (axioms), constructed using extra-logical symbols and the logical operators, which can be evaluated to either true or false.

e.g.: P∧Q → R is a theory

Structure:

An assignment of a truth value (true or false) to each extra-logical symbol. Given an assignment, a theory can be evaluated to be true or false.

e.g.: P is true
Q is true
R is false
⇒ P∧Q → R is false

Unit 1: Theories and Schemas Slide Number 10

In conventional propositional logic, logical symbols are the logical operators (or connectives) "∧", "∨", "¬", "→". These have a fixed meaning given by the truth tables of the connectives. But in order to evaluate, via these tables, whether a formula is true or false, we need to assign a truth value to the extra-logical symbols in the formula.

The extra-logical symbols (i.e the propositional letters) do not have a fixed meaning. An assignment of truth values to the propositional letters is called a *structure*, or sometimes an *interpretation*. (Note that in your first year course of Logic, structures were also called "situations".)

Given a structure, we can then evaluate any formula written in that language, using the meaning of the propositional letters and the truth tables for calculating the meaning of the complex formulae. If this calculation (i.e. evaluation) makes a given formula true, then we say that the formula is true in that structure, and that this chosen structure (interpretation) is a *model* of that formula.

Course: Software Engineering – Design I

First-order Logic

Language:

- logical operators: ∧, →, ¬, ∨, ∀, ∃ — have a fixed meaning
- extra-logical symbols: — don't have a fixed meaning
  - Constants
  - Function symbols
  - Predicate symbols

Constants refer to objects in a given domain:
e.g. **a** refers to Peter, if the domain is a set of people

Functions map (tuples of) objects to objects in a domain
e.g. **-(5,7)** refers to a minus operation over the integer numbers (Z)

Predicates describe relations on (tuples of) objects in a given domain
e.g. **brother(John, Susan)** refers to a brotherhood relation

In first-order logic, the language is much richer. Logical symbols include not only logical connectives, but also quantifiers (existential and universal) over variables. Extra-logical symbols include propositional letters, constants, variables, function symbols (with different arities) and predicate symbols (with different arities).

Constants are used to refer to individual elements of an underlying domain of discourse, function symbols are used to express function-like elements, such as the subtraction function between two integers, and predicate symbols are used to express relation-like concepts, e.g. the relation of brotherhood between people. Variables are, in our case, always assumed to be within the scope of a quantifier.

First-order Logic (continued)

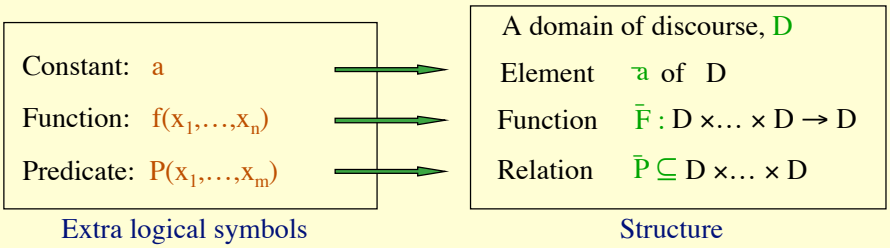
Theory:

Set of logical formulae (axioms), constructed using extra-logical symbols and logical operators, which can be evaluated to either true or false.

e.g.: $\exists x.\ \forall y.\ (P(a,y) \rightarrow Q(f(y),x))$ is a theory

Structure:

Define the meaning of each extra-logical symbol. It includes:

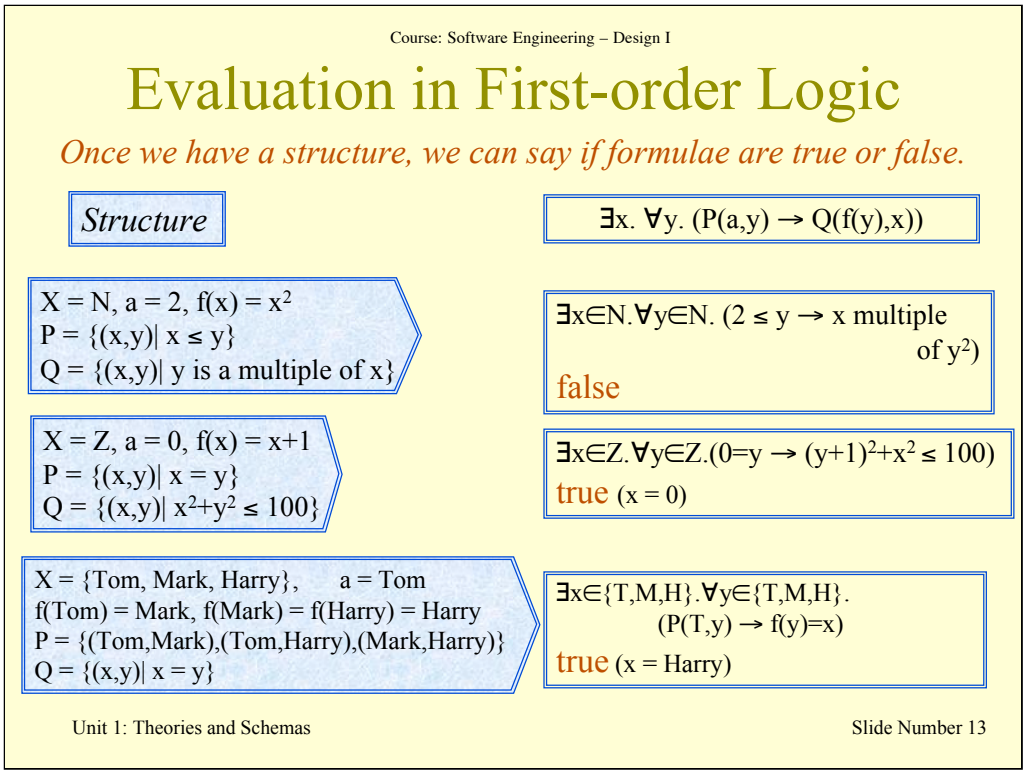


Extra logical symbols — Structure

Also in the case of first-order logic, a structure assigns meanings to the extra-logical symbols. Variable symbols range over a domain, also called a carrier, which can be thought of as a domain of discourse. Values of variables are elements of this domain. Constant symbols are interpreted in this domain as well. Functions are mapped into functions defined from the domain to itself. Cartesian products of the domain need to be considered as domains of those functions that have arity greater than one. Predicate symbols are instead mapped into subsets of (Cartesian products of) the domain.

Given a theory, we call "signature" the set of extra-logical symbols used in the theory. In the example theory above, the signature is given by the constant "a", the function "f", the binary predicates "P", and "Q".

Again, given a structure (i.e a meaning for all extra-logical symbols in the language), first-order formulae can be *evaluated* true or false within this structure. Let's see some examples of structures and evaluations of first-order formulae.



We consider here some examples of structures for the first-order theory given by the formula ∃x. ∀y. (P(a,y) → Q(f(y),x)), where the extra-logical symbols are the constant "a", the predicate symbols "P" and "Q", and the function symbol "f". For each of these structures we evaluate if this formula is true or false.

The structures provide meaning to the extra-logical symbols in set-theoretic terms. Once this meaning is fixed, the truth or falsity of a formula is given by the truth or falsity of its interpretation in the given structure. On the left-hand side of the slide, we have three alternative structures, on the right-hand side of the slide the interpretation of the given formulae in each of these structures. As we said before, structures that make a formula true are **models** for that formula and structures that make a formula false are not models.

The first structure makes the formula false. In the second structure, a model for the formula is given by taking x=0, whereas in the third structure a model is given by taking x=Harry.

Signature of a Theory

Given a theory, a signature is the set of extra-logical symbols used in the theory:

theory

$\exists x.\ \forall y.\ (P(a,y) \rightarrow Q(f(y),x))$

signature

$P \subseteq D \times D;\ Q \subseteq D \times D;\ a;$
$f: D \rightarrow D$

Models of a Theory

Given a set of formulae, or a theory, written in a given signature models are those structures that make each formula in the theory true.

Let's us emphasis once more two main concepts. Given a theory a signature is the set of extra-logical symbols that appear in the theory. Given a theory and its associated signature, a model is a structure for all the symbols in the signature which makes the formulae of the theory true.

Course: Software Engineering – Design I

Signatures, Theories, Structures, Models

- A ***signature*** (or *vocabulary* or *language*) describes the extra-logical ingredients that need to be interpreted.
- A ***theory*** comprises a signature and some logical formulae (axioms) constructed using the signature's symbols.
- Signatures are interpreted using ***structures*** (in which the ingredients are interpreted set-theoretically).
- Theories are interpreted using ***models*** (structures in which the axioms are true).

Unit 1: Theories and Schemas Slide Number 15

To summarise this review part of logic, the basic concepts in standard classical logic, are *signatures*, *theories*, *structures* and *models*.

A signature is the set of extra-logical symbols (i.e. vocabulary or language) that is used to describe a problem. These symbols need to be interpreted. Signatures are interpreted using structures, where all the symbols of the signature are interpreted set-theoretically.

A theory specifies a set of logical formulae (axioms), constructed using the symbols of a given signature. Therefore, a theory comprises both a signature and a set of formulae written in that signature. Theories are interpreted using models, i.e. those structures that makes the axioms of the theory true.

To draw an analogy between these formal concepts and our basic topic of formal specifications, logical theories will be for us formal specifications of a system behavior, whereas system behaviors are "real-world" models of our logical theory.

Syntax	Semantics
Signature →	Structure
Theories →	Models

Schemas: Basic Idea

- "Mathematical modelling" – describe a theory of which system should be model.
- Schemas specify system by describing its basic features and assumptions.
- System is to be a "real-world model" of the schema (e.g. domain of discourse are collections of objects in real world, not mathematical sets).
- Logical inference predicts properties of system.

In the introduction lecture distributed together with the notes, it is mentioned that a formal specification is a mathematical model that describes the system behaviour; "specifying" is therefore the process of "mathematically modelling" our system. Such a process gives us a logical theory, i.e. our formal specification, of which the system should be a model.

I have also mentioned in the introduction lecture notes that building a mathematical model of a system's behaviour involves focusing on some main aspects of the system, leaving out inessential details. These aspects are *basic features* and *assumptions*. Logical theories, or schemas, specify a system by describing its basic features (e.g., specific variables, functions symbols, predicates, etc.) and assumptions (e.g., axioms about the predicates).

Systems that are often specified, are more then just small programs with few variables. They are "real-world models" of schemas. Domains of discourse (or carriers) in the schemas are often collections of objects in the real world, not necessarily mathematical sets.

Logical inferences on defined schemas can be used to predict (or infer) properties of the systems. These properties are formulae inferred from the assumptions in the schema.

Schemas

We shall use ***schemas*** as notation for theories. A schema has two parts: signatures and axioms.

Theory name [carrier name]	
Signature	(in Z: *declarations)*
Axioms	(in Z: *predicate)*

- Schema notation is adapted from Z.
- Our logical view of schemas is different from that of Z
  - but their use in practice is very similar.

In the next few slides we will describe, and illustrate with examples, how the notion of a logical theory can be re-formulated using a **Z** notation called ***schema***.

Schemas in Z.

The **Z** specification language includes two types of notation: the axiomatic description and the schema notation. The first is mainly used to declare the types of the variables used in a specification. Axiomatic descriptions, also called axiomatic definitions, correspond to the declarative part of a program: they specify the types of the variables and some constraints on their possible values, but not their actual value. The schema is, instead, a notation used to model the "states" of a system, in terms of state variables and their values. Operations on schemas can be used to formalise system operations. A schema has a name and is a "closed box" to denote that the variables defined inside are *local* to that schema.

Our use of schemas.

In this course we see schemas as (logical) theories. So, a schema includes two main parts, the top part, above the line, which describes the signature of a logical theory, the bottom part, below the line, which describes the axioms of the theory. Each schema is identified by its own "*name*", which is also the theory name. Next to the schema's name, we can specify the domain of concern, i.e. the (real-life) domain the theory is referring to. Some examples are given in the additional set of notes. We can call the domain of discourse "carrier".

Comparing the two.

Our logical view of schemas is not exactly the same as that used in Z, e.g., the specification language Z includes also special types which are not definable in pure first-order logic. However, our use of schemas is in practice similar to that of Z. For instance, in standard Z definition of a schema, the signature part is called the declaration and is used to define the variables used within the schema and their types. The axioms part is called the predicate and includes constraints that restrict the variables' values.

Terminology

- *Vocabulary*, *signature*, *extra-logical symbols* and (in Z) *declaration* all mean more or less the same.
- So do *assumptions*, *premisses*, *axioms* and (in Z) the *predicate*.
- "Predicate" in Z is not the same as "predicate" in first-order logic.

Before seeing some examples of logical theories as schemas, it is necessary to make some small remarks about the terminology that we are going to used in the rest of this course.

We have just to remember that:

1) "Vocabulary", "signature", "extra-logical symbols" mean more or less the same thing. In Z the term "declaration" is used instead.

2) Assumptions and axioms also mean the same thing. Z uses the term "predicate" instead.

Be careful: The term "predicate" in Z is not the same as in logical theories. They denote two different things. What? You should know the answer by now!

Course: Software Engineering – Design I

Example: a logical theory as schema

- A schema can be used to describe vocabulary and axioms.
  Consider the logical theory defined by:

$$\exists x.\ \forall y.\ (P(a,y) \rightarrow Q(f(y),x))$$

The schema

Prems [X]

$a: X$

$f: X \rightarrow X$

$P, Q \subseteq X \times X$

$\exists x{:}X.\ \forall y{:}X.\ (P(a,y) \rightarrow Q(f(y),x))$

- "Prems" is just a name we've invented for this schema
- Given a schema, we can draw *inferences* from it
  - logical consequences of the axioms using the symbols in the signature.

Unit 1: Theories and Schemas Slide Number 19

Schemas can be used to describe the signature and the axioms that form a logical theory. In this example we consider, for instance, the logical theory defined by the formula:

$\exists x.\forall y.(P(a,y) \rightarrow Q(f(y),x)))$

Signature: { P, Q, a, f, x, y }, where "P" and "Q" were binary predicates, "a" is a constant symbol, "f" a function symbol and "x" and "y" two bounded variables.

Axioms: $\{\exists x.\forall y.(P(a,y) \rightarrow Q(f(y),x))\}$.

We have also seen different structures where the domain of interpretation, X, was either the set of natural numbers, or the set of integers or a specific set of people.

We can represent this theory as a schema in the following way. Define in the part above the line the signature of the theory, except the bounded variable symbols. In the part below the line include the axioms part of the theory. We then define a name for the schema, e.g. *Prems*, and specify the domain of discourse in square brackets, e.g. [X].

Given a logical theory, formal proofs can be constructed, which allow the inference of new formulae from the given axioms, using only the symbols in the given signature. For instance, it could be shown that from the premise $\exists x.\forall y.(P(a,y) \rightarrow Q(f(y),x)))$ it is possible to derive $\forall y.(P(a,y) \rightarrow \exists x.Q(f(y),x))$, which also uses the same signature. In the same way, given a schema, it is possible to draw inferences from it. Time permitted, we might briefly introduce the concept of derivation processes from schemas at the end of this six lectures.

What is important to learn at the moment is that, as far as formal specifications of systems is concerned, logical theories are formal specifications, systems are real-world models of the specification, and reasoning processes can be used on formal specifications to infer properties of the system.

Notation

Prems [X]

a: X
f: X → X
P, Q ⊆ X×X

∃x:X. ∀y:X. (P(a,y) → Q(f(y),x)))

- ":" means "of type"
- "→" means functions - so f: X → X means f is a function with one argument
- "⊆" is used for predicates
- "×" means Cartesian product - so P ⊆ X×X means P is a predicate with two arguments
- The *bound* variables, x and y, don't need declaring in the signature.

These are some of the notation rules used to write logical theories as schemas.

The elements of a signature need to have their types declared. To specify the type, we use the ":" symbol.

Constant symbols of a schema need to be declared individually with their respective types. (We will see later on that it is possible to specify constant symbols of different types in a logical theory and therefore in a schema.) In this case variables and constant symbols are of only one type, X, left unspecified , e.g. they could be of type natural numbers if for instance X is given by N.

Function symbols need to be declared to be functions, i.e. using the symbol "→" and specifying the domain and co-domain. This type declaration also states the arity of the function symbol. In this example, f is a function with one argument.

Predicate symbols are declared using the set inclusion ⊆ symbol. Their arity is defined in terms of the Cartesian product of the type(s) of its arguments. The symbol for Cartesian product is ×.

Variables that are quantified, i.e. bound variables, do not need to be declared in a schema.

Hence, the signature of a schema defines the signature of a logical theory in terms of (extra-logical) symbols and their respective types; the axioms of a schema define the axioms of a theory.

Example of a schema

S: N → N
S(0) = 0 ∀x:N. S(x+1) = S(x)+x+1

- N is a **special purpose set** with its own operators, predicates and reasoning principles already defined.
- No carrier needed!

  Structure needs carrier to show "domain specific" range of variation of variables - but in this case, the carrier is fixed for variables of type N.
- No need to declare 0 or + in the signature.

This is an example of a schema that specifies the mathematical concept of induction over natural numbers.

The signature includes the constants 0 and 1, the predicate "=" ("=" $\subseteq$ N×N), the function symbol S, which takes a natural number and gives a new natural number (after performing some operations), and the function symbols +. The domain is the set of natural numbers N.

The set N is a **special purpose set**, which comes with its own operators (e.g., + and *), predicates (e.g., =) and reasoning principles such as induction. These are already defined together with N.

Because N is a special purpose domain, it does not need to be defined as the carrier of the schema. The definition of a carrier in a schema is mainly to show system dependent domains (i.e. range of values for the variables). Once we know that the variables are natural numbers, we know already the possible values that they can assume. So this schema does not need any carrier.

What about its signature? If we eliminate from the signature of the theory, the symbols which are already defined within the domain of natural numbers, the only extra-logical symbol that needs to be declared is just the function S. This is a function from N → N, as shown in the picture.

The axioms part of the schema includes the two axioms of the theory.

So, to summarise, when we define a schema, carriers are only used to specify particular (system dependent) sets of values for the variables.

Let's consider another example.

A list example

ScrubFn [X]

scrub: X × seqX → seqX

∀y:X. scrub(y,[ ]) = [ ]
∀y,x:X. ∀xs: seqX.
  (y≠x → scrub(y, x:xs) = [x] ++ scrub(y,xs))
∀x:X. ∀xs: seqX. scrub(x, x:xs) = scrub(x,xs)

- seqX is the type of finite sequences (lists) from X
- ":" is sometimes "of type", sometimes "cons"
  - » you can tell which by the context
- ++ and […] are ordinary list notation
- "scrub" means "delete all instances of a given element from the list".

This is another example of a schema, where instead a carrier is needed and specified. This is because we want the schema to be general for any given set of elements. The schema describes a function, called "scrub", which eliminates from a given list of elements all the occurrences of a given element. The elements assume a value in the carrier X, and lists of elements are constructed from X and denoted by the constructed sort seqX.

The type seqX is therefore constructed from the carrier X. Again we have a special purpose domain which is a set of lists of elements. Given any domain of this type, we know already operators such as ++, for concatenation of lists, the symbol […] which denote a list, the symbol [] which denotes an empty list, and the function ":", called *cons* (for constructing lists), which can also be used to identify the head and tail of a list. All these elements are already known and defined, once a domain of type seqX is considered. Again, they don't need to be specified in the signature of a schema.

Let's work out how the function scrub is defined. (Done during the lecture).

Schema inclusion (an example)

Consider now the extended theory:

$\{ \exists x.\ \forall y.\ (P(a,y) \rightarrow Q(f(y),x)),$
$\forall y.\ (P(a,y) \rightarrow Q(f(y),x_0))),$
$P(a,y_0)\}$

ExtPrems

Prems[X]

x_0, y_0: X

$\forall y.\ (P(a,y) \rightarrow Q(f(y),x_0)))$

$P(a,y_0)$

In real world scenarios, it is possible that parts of a system to be specified can be defined in terms of other parts of the system already specified. We can use the notion of schema inclusion to capture this modularity when writing a formal specification. We present here the concept of schema inclusion in terms of logical theories. In the next slide, we will see the general definition of schema inclusion.

Let's consider again the theory given in slide 15. For simplicity we refer to this theory with the schema name Prems. Let's assume now that we want to express as a schema a new theory which extends Prems with additional two axioms. The new theory, referred to as ExtPrems, is then defined by the axioms $\{\exists x.\ \forall y.\ (P(a,y) \rightarrow Q(f(y),x)),\ \forall y.\ (P(a,y) \rightarrow Q(f(y),x_0))),\ P(a,y_0)\}$.

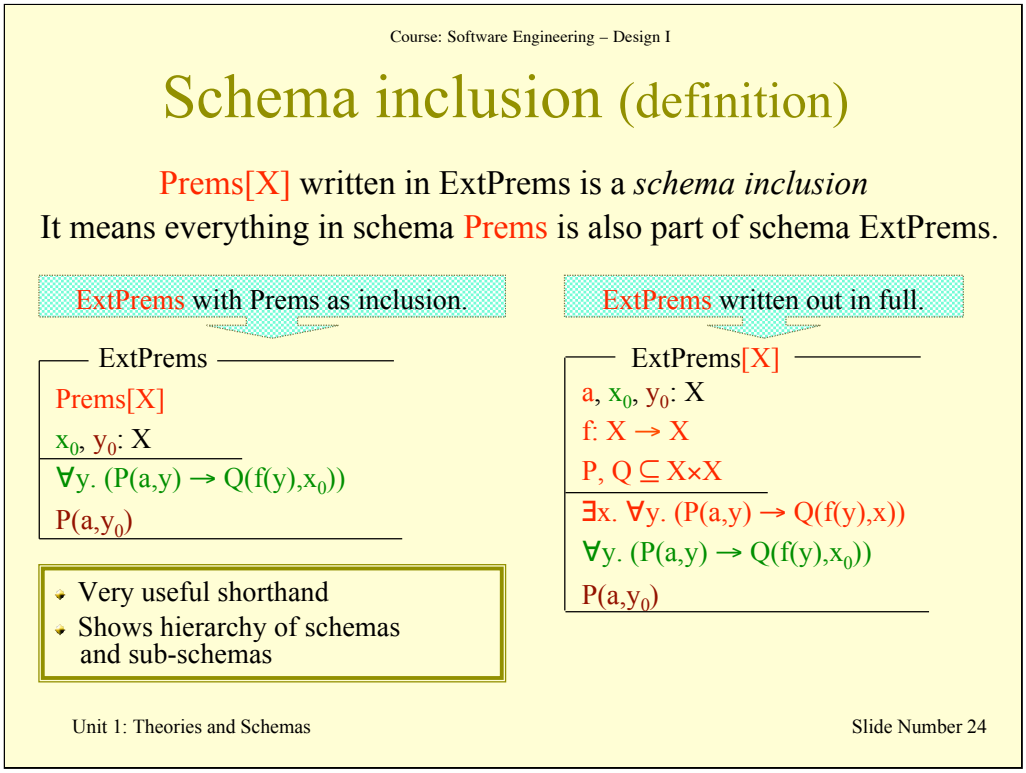
Note that, the two new axioms in ExtPrems use new symbols x_0 and y_0, respectively, which were not included in the signature of the theory Prems.

We can define the new theory ExtPrems with a new schema called "ExtPrems". This new schema includes the theory Prems, the new symbols x_0 and y_0, and the new two axioms. This is shown in the slide above. Note that there is no need to specify also the axioms of the initial theory, as these are already included in (the predicate part of) *Prems[X]*.

The operation of using an already defined schema inside the signature of another schema is called *schema inclusion*.

In the same way as the initial logical theory is included in the new theory, the initial schema *Prems[X]* is included in the new schema ExtPrems..

We define the schema inclusion operation in more detail in the next slide.



Schema inclusion is a shorthand notation for defining schemas. In the left-hand side of the slide, we have the definition of the schema *ExtPrems* which uses *Prems[X]* as a schema inclusion.

Prems[X] can be expanded within the definition of *ExtPrems*, writing *explicitly* the signature of *Prems[X]* in the signature of *ExtPrems* and writing explicitly the axioms of *Prems[X]* in the axiom part of *ExtPrems*. This gives the *expanded* definition of *ExtPrems*, which is in the right-hand side of the slide.

Schema inclusion is useful not only because it provides a shorthand notation, but also because it shows a hierarchy of schemas and sub-schemas, which turns out to be particularly useful in the specification of object oriented features.

Many-sorted Logic

Signature

A set of sorts

Sorts are like types in a programming language

A set of constant symbols, each with its own sort

A set of predicate symbols, each with a given arity

Arity is a finite list of sorts, e.g. Xs = [X1, …, Xn],
P is of this arity: P⊆ X1 × … × Xn

A set of function symbols, each with a given arity

Arity is a pair (Xs,Y) where Xs is a list of sorts and Y is a sort,
F is a function of this arity: F: X1 × … × Xn →Y

Formulae can be meaningless simply because they are not "well-typed".

Real-world systems often include parameters, or components, of different types. Specifications of large systems need therefore to take into account these differences. For instance, imagine that you are specifying the interaction between a system and its environment. Such a specification would therefore describe the overall process "environment-system". Environmental features and system features then constitute two different types (categories) of features in this overall process.

One way of formalising such specifications is by using many-sorted logic. This is a predicate logic whose signature is a many-sorted vocabulary, and as such, helps capture different categories of features of the underlying system.

A detailed definition of many sorted logic is given in the additional notes.

Structures for Many-sorted logic

Assume a given signature, then a **structure** for it comprises:

For each sort X,

- a corresponding set [X], or carrier (or domain) of X

For each constant a of sort X,

- an element $\bar{a}$ of [X],

For each predicate symbol $P \subseteq X1 \times \ldots \times Xn$,

- a corresponding subset $\bar{P}$ of the Cartesian product $[X1] \times \ldots \times [Xn]$

For each function symbol $f: X1 \times \ldots \times Xn \rightarrow Y$,

- a corresponding function $\bar{F}$ from $[X1] \times \ldots \times [Xn]$ to [Y].

Examples of many-sorted formulae

1) All IC students graduate with a first

$\forall s{:}StudentIC.\ (grade(s)=first)$

2) All employees whose salary is under 20000 pounds

$\forall e{:}Employees.\ (salary(e) < 20000).$

3) Italians read only Topolino

$\forall b{:}books, \forall p{:}people.\ [(italian(p) \wedge read(p,b)) \rightarrow b=topolino]$

4) All borrowed books are not in the library

$\forall b{:}Lib\_books.\ (borrowed(b) \rightarrow status(b) = out\_library).$

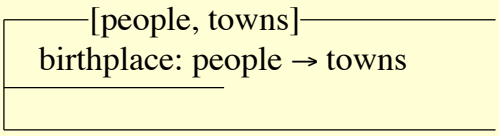
5) Students can borrow only one book

$\forall t{:}Time, \forall s{:}Students, \forall b,b1{:}Lib\_books.$

$(borrowing(s,b,t) \wedge borrowing(s,b1,t) \rightarrow b=b1).$

Many-sorted Logic Example of Schema

Consider this:



- Needs two carriers!
  - Structure = pair of sets with function between them.
- "people" and "towns" are two *sorts* in a *many-sorted* signature.
- One sort (ordinary predicate logic).
- No sorts (propositional logic).

The example above is a schema, with two sorts (i.e. two carriers) "people" and "town". The signature includes two carriers and a function symbol defined between these two carriers.

Note that : ordinary predicate logic is also, in effect, a particular many-sorted logic, with only one sort! In the same way we can see propositional logic as a particular many-sorted logic with zero sorts!

Sorts and types

- Sorts (logic) and types (programming) are roughly the same.
- Can construct lots, e.g. N, seqX, etc.
- The only ones that go in square brackets at the top of the schema are the "primitive sorts", which are not special purpose sets or sets constructed out of others.

right

[X]

a:N etc.

…..

wrong

[X,N,seq X]

a:N etc.

…..

You can consider sorts in logic as types in programming languages. They are roughly the same thing. Functions and predicates can be defined on different sorts. The arities of predicates and function symbols are used to specify the sorts of their arguments and (in the case of functions) their results. So the arity of a predicate P is a finite list of sorts, (e.g., Xs= $[X_1,\ldots,X_n]$). Then P can be used only with n arguments, $P(x_1,..x_n)$, where each argument x_i has to belong to its associated sort X_i. Similarly for functions. The arity of a function is a pair (Xs, Y), where Xs is a list of sorts defining the number and sorts of its arguments and Y is the sort of the function value. Formulae are "well-typed" if the use of arguments of predicates and functions respects their sorts! This is similar to the idea of "well-typed" expressions in programming languages.

A many-sorted theory can include any number (starting from zero) of sorts. Structures for a many-sorted signature therefore include, for each sort, an associated domain or carrier.

Sorts can be constructed from other sorts. For instance, given a sort X, we can construct the sort seq X, or the sort **F**X of finite power set of X. When a signature includes constructed sorts like this, a structure for it must have its carriers constructed in the corresponding way.

In the schema representation, however, only the ***primitive sorts*** need to be specified, as the other constructed sorts can be automatically and unambiguously defined. Examples of correct and wrong schema definitions are given here.

Summary

- A schema is a way of describing a *logical theory*.
- The description has
  - sorts (primitive sorts, constructed sorts like N, seqX)
  - constants, functions, predicates, propositions
  - axioms.
- A schema inclusion is a shorthand notation for schemas.
- A schema has models.
- If the schema is a specification, then a model is a system implementation that satisfies it.

To summarise, a logical theory comprises a signature and some axioms. **Signatures are interpreted in structures, where each extra-logical symbol finds its own meaning. Models are those structures that give interpretations to the signature in such a way as to make the axioms of the theory hold. So we can say the following.**

A schema is a way of describing a logical theory. It includes a name, sorts (the base sorts are specified in square brackets next to the name), and symbols of the theory's signature, such as constants, functions, predicates. The axioms of the theory are also specified in the schema. Large schemas can be shortened by using the notation of schema inclusion.

Schemas have models in the same way as theories have models.

Now, if we consider a schema as a formal specification, the implemented system specified by the schema can be seen, in effect, as a model of the schema, in the sense that the implemented system has to satisfy or meet the schema specifications.

In Unit 2 we are going to see how schemas can be used to describe systems' behaviours. To do so we will need to define two main notions: **state schemas** and **operation schemas**. The additional notes distributed at this lecture include additional material on the definition of first-order and many-sorted logics as well as on the topics that we are going to cover in the remaining lectures.