

System Modelling and Design

COMP2111



Johannes Åman Pohjola

Formal

System Modelling and Design COMP2111



Johannes Åman Pohjola

We'll learn to

model systems in a way that's unambiguous and mathematically precise.

We'll be able to

say what it means for a system to satisfy its specification,
and prove that it does so.

We'll need

a substantial toolbox of discrete math and formal logic.

Don't worry; we'll teach it, not assume it.

We'll learn to

model systems in a way that's unambiguous and mathematically precise.

We'll be able to

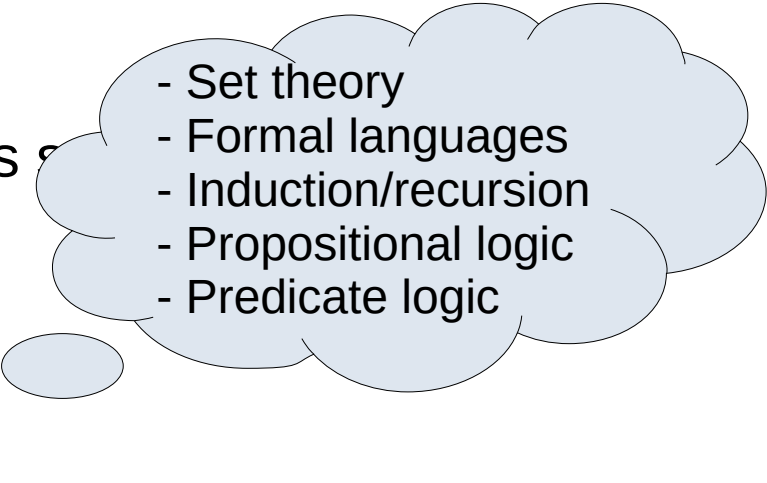
say what it means for a system to satisfy its spec

and prove that it does so.

We'll need

a substantial toolbox of discrete math and formal logic.

Don't worry; we'll teach it, not assume it.

- 
- Set theory
 - Formal languages
 - Induction/recursion
 - Propositional logic
 - Predicate logic

We'll learn to

model systems in a way that's unambiguous and mathematically precise.

We'll be able to

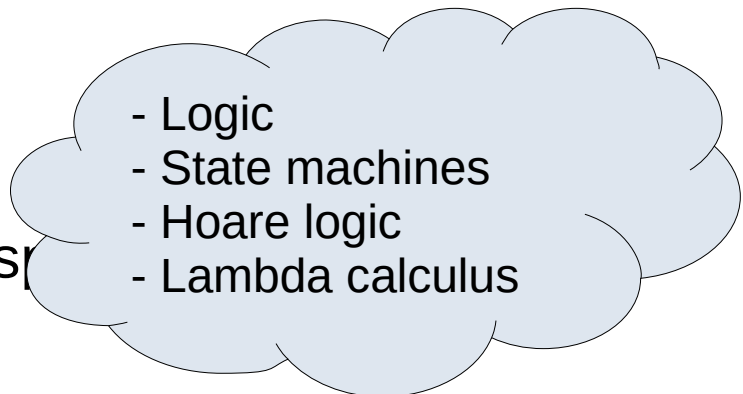
say what it means for a system to satisfy its spec

and prove that it does so.

We'll need

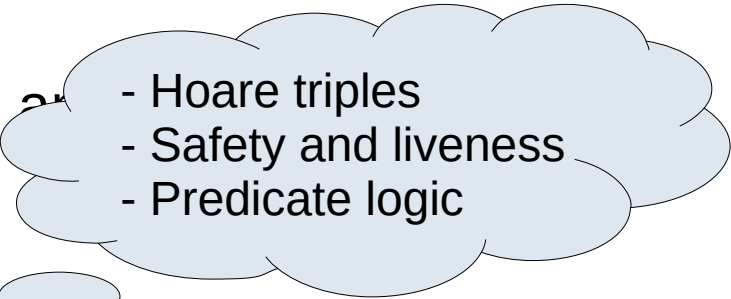
a substantial toolbox of discrete math and formal logic.

Don't worry; we'll teach it, not assume it.

- 
- Logic
 - State machines
 - Hoare logic
 - Lambda calculus

We'll learn to

model systems in a way that's unambiguous and mathematically precise.

- 
- Hoare triples
 - Safety and liveness
 - Predicate logic

We'll be able to

say what it means for a system to satisfy its specification,

and prove that it does so.

We'll need

a substantial toolbox of discrete math and formal logic.

Don't worry; we'll teach it, not assume it.

We'll learn to

model systems in a way that's unambiguous and mathematically precise.

We'll be able to

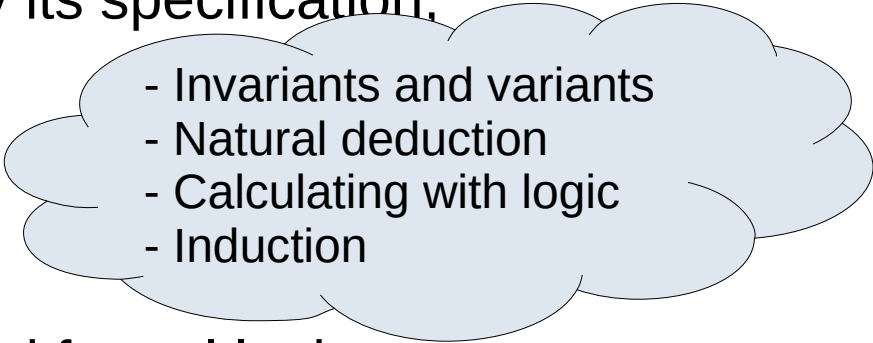
say what it means for a system to satisfy its specification,

and prove that it does so.

We'll need

a substantial toolbox of discrete math and formal logic.

Don't worry; we'll teach it, not assume it.

- 
- Invariants and variants
 - Natural deduction
 - Calculating with logic
 - Induction

Non-examples

September 1981

Transmission Control Protocol
Functional Specification

3.9. Event Processing

The processing depicted in this section is an example of one possible implementation. Other implementations may have slightly different processing sequences, but they should differ from those in this section only in detail, not in substance.

The activity of the TCP can be characterized as responding to events. The events that occur can be cast into three categories: user calls, arriving segments, and timeouts. This section describes the processing the TCP does in response to each of the events. In many cases the processing required depends on the state of the connection.

Events that occur:

User Calls

OPEN
SEND
RECEIVE
CLOSE
ABORT
STATUS

Arriving Segments

SEGMENT ARRIVES

Timeouts

USER TIMEOUT
RETRANSMISSION TIMEOUT
TIME-WAIT TIMEOUT

The model of the TCP/user interface is that user commands receive an immediate return and possibly a delayed response via an event or pseudo interrupt. In the following descriptions, the term "signal" means cause a delayed response.

Error responses are given as character strings. For example, user commands referencing connections that do not exist receive "error: connection not open".

Please note in the following that all arithmetic on sequence numbers, acknowledgment numbers, windows, et cetera, is modulo 2^{32} the size of the sequence number space. Also note that " $=<$ " means less than or equal to (modulo 2^{32}).

Non-examples

This RFC is a specification in English.

Natural language specs tend to have:

- Ambiguities
- Room for interpretation
- Important details in the writer's head absent from actual text.

September 1981

Transmission Control Protocol
Functional Specification

3.9. Event Processing

The processing depicted in this section is an example of one possible implementation. Other implementations may have slightly different processing sequences, but they should differ from those in this section only in detail, not in substance.

The activity of the TCP can be characterized as responding to events. The events that occur can be cast into three categories: user calls, arriving segments, and timeouts. This section describes the processing the TCP does in response to each of the events. In many cases the processing required depends on the state of the connection.

Events that occur:

User Calls

OPEN
SEND
RECEIVE
CLOSE
ABORT
STATUS

Arriving Segments

SEGMENT ARRIVES

Timeouts

USER TIMEOUT
RETRANSMISSION TIMEOUT
TIME-WAIT TIMEOUT

The model of the TCP/user interface is that user commands receive an immediate return and possibly a delayed response via an event or pseudo interrupt. In the following descriptions, the term "signal" means cause a delayed response.

Error responses are given as character strings. For example, user commands referencing connections that do not exist receive "error: connection not open".

Please note in the following that all arithmetic on sequence numbers, acknowledgment numbers, windows, et cetera, is modulo 2^{32} the size of the sequence number space. Also note that " $=<$ " means less than or equal to (modulo 2^{32}).

Non-examples

This RFC is a specification in English.

Natural language specs tend to have:

- Ambiguities
- Room for interpretation
- Important details in the writer's head absent from actual text.

You've experienced this yourself
now, in Assignment 2.

3.9. Event Processing

The processing depicted in this section is an example of one possible implementation. Other implementations may have slightly different processing sequences, but they should differ from those in this section only in detail, not in substance.

The activity of the TCP can be characterized as responding to events. The events that occur can be cast into three categories: user calls, arriving segments, and timeouts. This section describes the processing the TCP does in response to each of the events. In many cases the processing required depends on the state of the connection.

Events that occur:

User Calls

OPEN
SEND
RECEIVE
CLOSE
ABORT
STATUS

Arriving Segments

SEGMENT ARRIVES

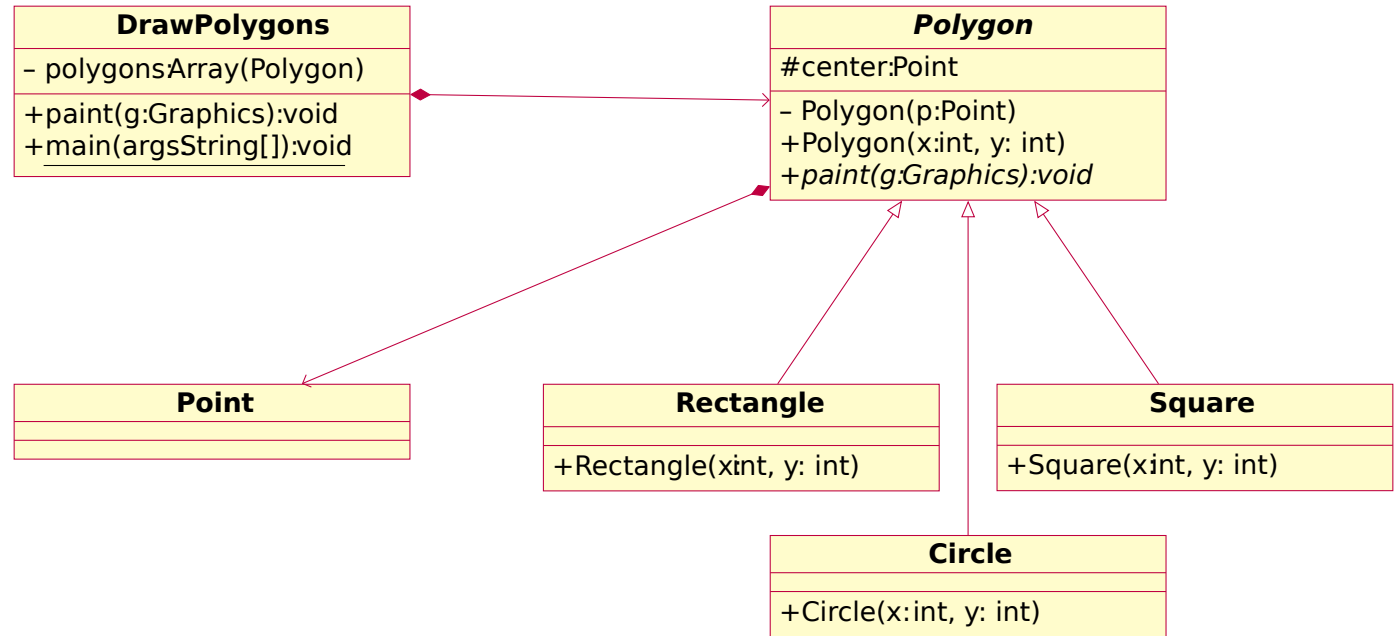
Timeouts

Commands receive an
an event or
term "signal"

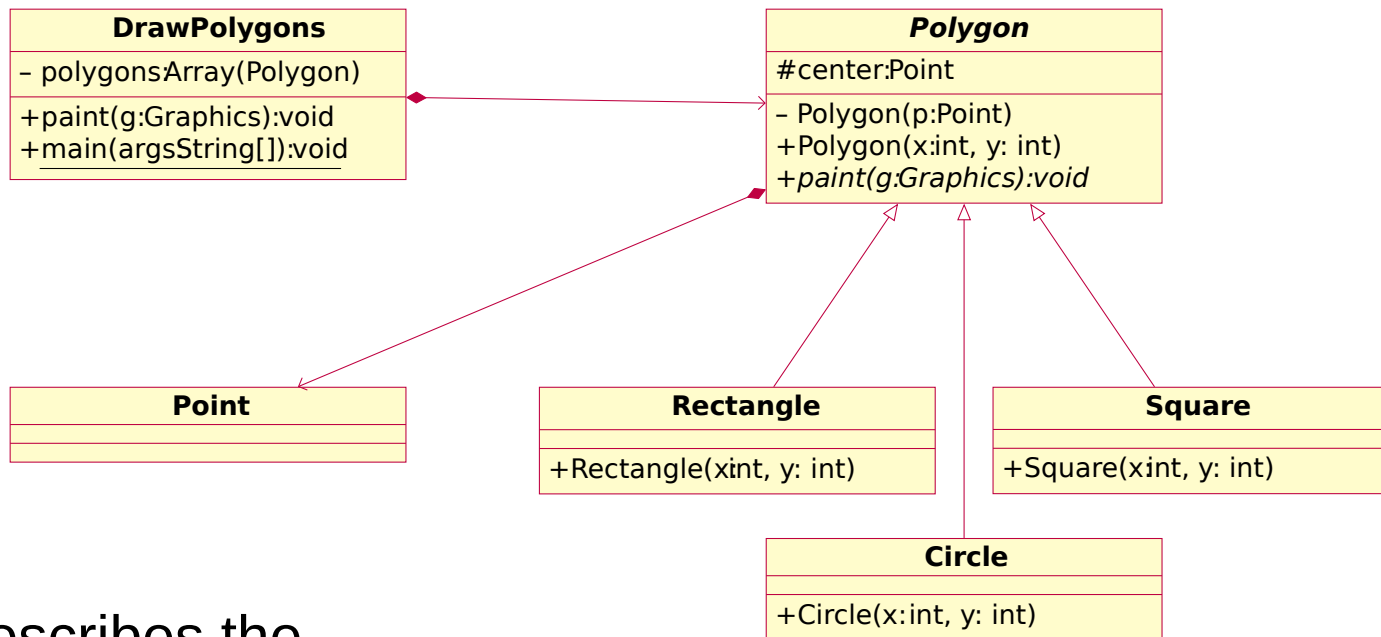
For example, user
not exist receive "error:"

The following that all arithmetic on sequence numbers,
segment numbers, windows, et cetera, is modulo 2^{32} the size
of the sequence number space. Also note that " $=$ " means less than or
equal to (modulo 2^{32}).

Non-examples



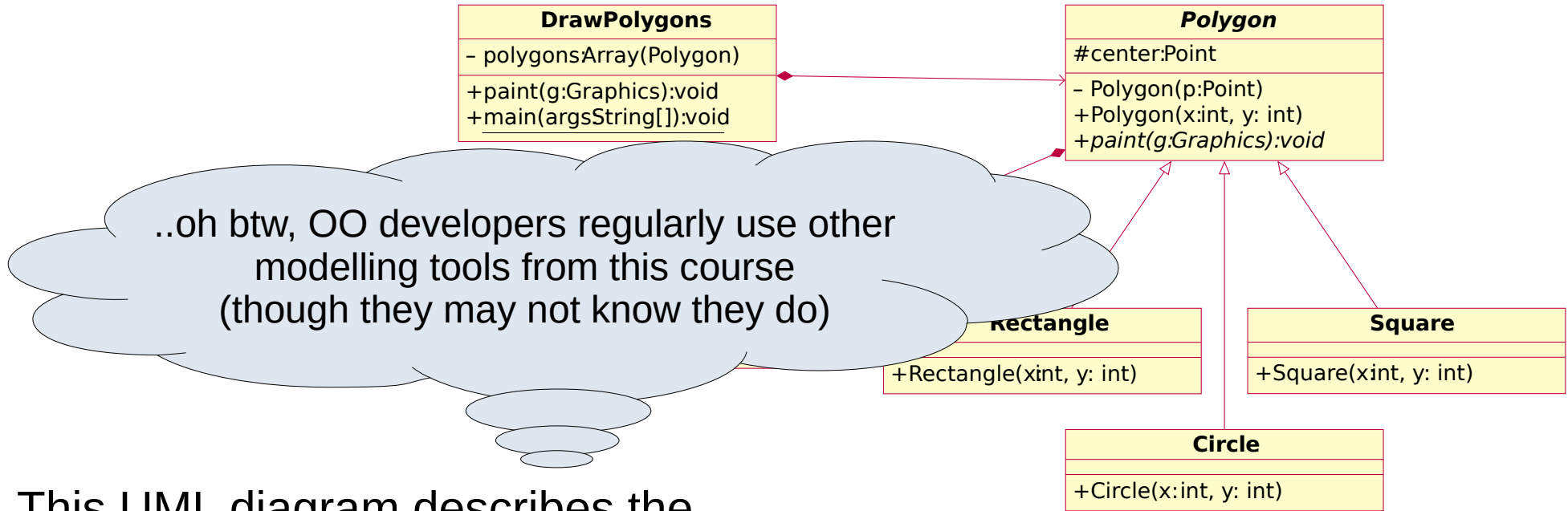
Non-examples



This UML diagram describes the *structure* of the system, not its behaviour.

Could we *give* it meaning using the tools from this course?

Non-examples



This UML diagram describes the *structure* of the system, not its behaviour.

From Java's library documentation

max

```
public static <T extends Object & Comparable<? super T>> T max(Collection<? extends T> coll)
```

Returns the maximum element of the given collection, according to the *natural ordering* of its elements. All elements in the collection must implement the `Comparable` interface. Furthermore, all elements in the collection must be *mutually comparable* (that is, `e1.compareTo(e2)` must not throw a `ClassCastException` for any elements `e1` and `e2` in the collection).

This method iterates over the entire collection, hence it requires time proportional to the size of the collection.

Type Parameters:

T - the class of the objects in the collection

Parameters:

coll - the collection whose maximum element is to be determined.

Returns:

the maximum element of the given collection, according to the *natural ordering* of its elements.

Throws:

`ClassCastException` - if the collection contains elements that are not *mutually comparable* (for example, strings and integers).

`NoSuchElementException` - if the collection is empty.

See Also:

`Comparable`

From Java's library documentation

max

```
public static <T extends Object & Comparable<? super T>> T max(Collection<? extends T> coll)
```

Returns the maximum element of the given collection, according to the *natural ordering* of its elements. All elements in the collection must implement the `Comparable` interface. Furthermore, all elements in the collection must be *mutually comparable* (that is, `e1.compareTo(e2)` must not throw a `ClassCastException` for any elements `e1` and `e2` in the collection).

This method iterates over the entire collection, hence it requires time proportional to the size of the collection.

Type Parameters:

T - the class of the objects in the collection

Parameters:

coll - the collection whose maximum element is to be determined.

Returns:

the maximum element of the given collection, according to the *natural ordering* of its elements.

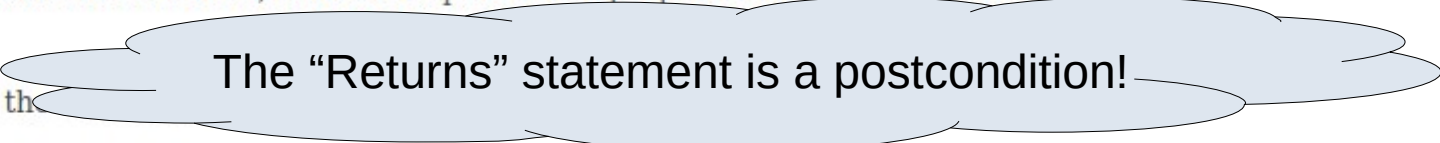
Throws:

`ClassCastException` - if the collection contains elements that are not *mutually comparable* (for example, strings and integers).

`NoSuchElementException` - if the collection is empty.

See Also:

`Comparable`



The "Returns" statement is a postcondition!

From Java's library documentation

max

```
public static <T extends Object & Comparable<? super T>> T max(Collection<? extends T> coll)
```

Returns the maximum element of the given collection, according to the *natural ordering* of its elements. All elements in the collection must implement the `Comparable` interface. Furthermore, all elements in the collection must be *mutually comparable* (that is, `e1.compareTo(e2)` must not throw a `ClassCastException` for any elements `e1` and `e2` in the collection).

This method iterates over the entire collection, hence it requires time proportional to the size of the collection.

Type Parameters:

T - the class of the objects in the collection

Parameters:

coll - the collection

Returns:

the maximum element of the collection, according to the *natural ordering* of its elements.

Throws:

`ClassCastException` - if the collection contains elements that are not *mutually comparable* (for example, strings and integers).

`NoSuchElementException` - if the collection is empty.

See Also:

`Comparable`

The “Throws” statement is a (negated) precondition!

From Java's library documentation

max

In OO design this is called *design by contract*, not Hoare triples, but it's the same thing.

You know how to *prove* this function is correct: Hoare logic.

Formal methods have found bugs in Java's standard library before.

Maybe you'll find the next bug? :)

`coll` - the collection whose maximum element is to be determined.

Returns:

the maximum element of the given collection, according to the *natural ordering* of its elements.

Throws:

`ClassCastException` - if the collection contains elements that are not *mutually comparable* (for example, strings and integers).

`NoSuchElementException` - if the collection is empty.

See Also:

`Comparable`

Practice exam

Exam coming up soon: May 9th 8AM - May 10th 8AM

The 2021 exam is available for practice on the course website.

It's mostly representative of what you should expect. Any topic covered in the lectures and tutes may come up.

Related courses

COMP3151: Foundations of Concurrency
COMP3161: Concepts of Programming Languages
COMP3153: Algorithmic Program Verification
COMP4141: Theory of Computation
COMP4161: Advanced Topics in Software Verification
COMP9020: Foundations of Computer Science
COMP6721: (In-)formal Methods
SENG2011: Workshop on Reasoning About Programs

That's all folks!

Thanks for attending the course.

Please don't forget to do the myExperience survey.

Come talk to me if you're interested in thesis/ToR projects :)