Teaching Unifying Theories of Programming

From semantic meta-theory to refinement literacy

Jim Woodcock

Southwest University | Aarhus University | University of York

FMTea 2026, Tokyo

19 May 2026

ChatGPT produced the pictures

Unifying Theories of Programming in a nutshell

Horizontal axis Link different paradigms: compare and contrast.

Vertical axis Link different levels of abstraction: refinement.

z-axis Link semantic presentations: denotational, algebraic, operational.

Family axis Link theories through parametrisation.

UTP Insights for Students

Lessons

1. Programs are characterised by observations.
2. Programs can be treated uniformly as predicates.
3. Healthiness conditions define the programming theory.
4. Refinement is the central ordering principle.
5. Different semantic styles can be connected.
6. Theories are built compositionally.
7. Semantic differences explain law differences.
8. UTP separates syntax from semantics.
9. It gives a theory of semantic engineering.
10. It gives us unification without flattening difference.

Teaching Unifying Theories of Programming

UTP is difficult

UTP becomes teachable when it becomes a method.

Not first

Catalogue of definitions.

But first

Disciplined way to progress.

Goal

Refinement literacy.

Thesis

- Issue isn't whether students can eventually read definitions.
- Issue is whether they can act when faced with a specification.
- Students should know what to do next.
- Calculate, simplify, apply a law, record side conditions, justify refinement.

UTP is simple in principle, but . . .

UTP is difficult because it asks students to stop treating semantics as a hidden explanation of a language, and start treating semantics as an explicit design object.

What students find difficult

1. Students must learn to think semantically, not syntactically.
2. Theory observations are unfamiliar.
3. Healthiness conditions are powerful but abstract.
4. Refinement reverses ordinary implication.
5. UTP unifies too many things at once.
6. There are several semantic presentations.
7. The notation looks elementary but isn't.
8. The examples can become too abstract. A theory about theories.
9. Tool support changes the teaching problem.
10. The main idea is meta-theoretical

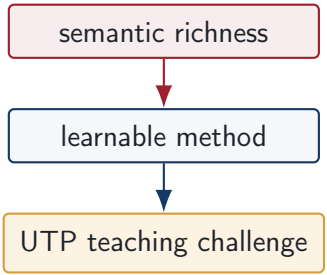
The teaching problem

Real systems need rich semantics

Concurrency, interaction, partiality, time, probability, hybrid behaviour, and implementation detail.

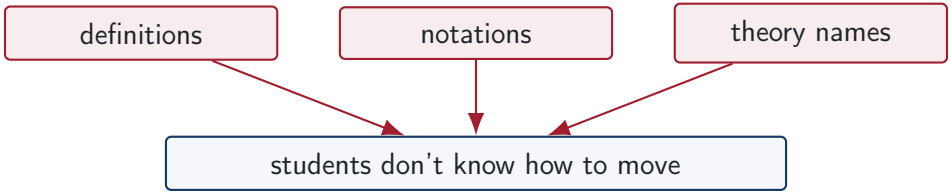
Students need a clear next step

A small number of stable ideas, practised repeatedly, with fast feedback.



- Formal methods teaching always balances realism and learnability.
- UTP is rich, but teaching must reveal the method.

What goes wrong when we teach UTP badly



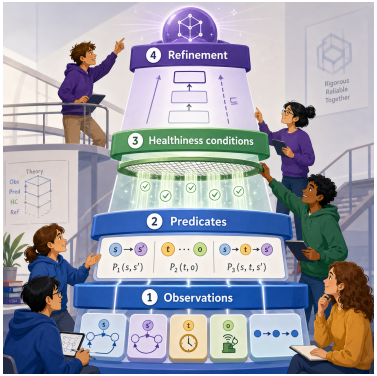
The problem

- The symptom isn't ignorance. It's blocked calculation.
- Main diagnosis. What do you do with definitions?
- The block is a pedagogical failure, not a mathematical one.
- Formal methods isn't a spectator sport (after Pólya).
- Student have to use the notations, ideas, and methods.

The UTP teaching manifesto

We get a UTP programming theory by

1. Choose observations and alphabets.
2. Define predicates over them.
3. Impose healthiness conditions.
4. Order the result by refinement.
5. Link to other theories.
6. Explore the semantics.



Student map: What to observe, what counts as behaviour, what counts as well-formed, and how programs improve. This is the pedagogical version of the UTP thesis: it's memorable, operational, and reusable.

Three unifications to teach explicitly

Across theories
relations
designs
reactive designs
processes

Across abstraction
requirements
contracts
code
machine

Across presentations
denotational
algebraic
operational
axiomatic

The pedagogical claim

Students need to traverse all three dimensions, but not all at once.

These are the x, y, and z axes of the teaching programme.

Roadmap

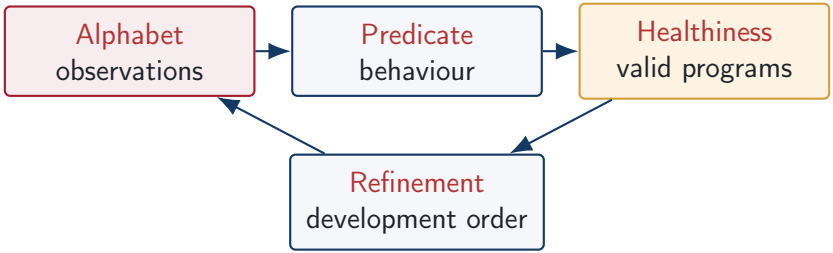
The semantic core

From concepts to a course

Feedback, tools, and artefacts

The call to action

UTP as a semantic compass



Teach students to ask these four questions on every example.
It's useful because it applies equally to relations, designs, reactive contracts, and process algebras.

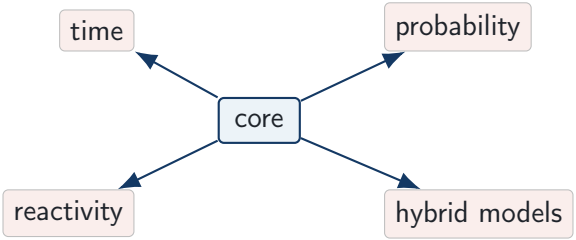
The minimal teachable core

Topics

1. Alphabetised relational calculus
2. Refinement and lattice structure
3. Healthiness conditions
4. Linking constructions

Teaching discipline

- Teach the core early.
- Add reactivity, time, probability, hybrid behaviour, and mechanisation later as controlled extensions.



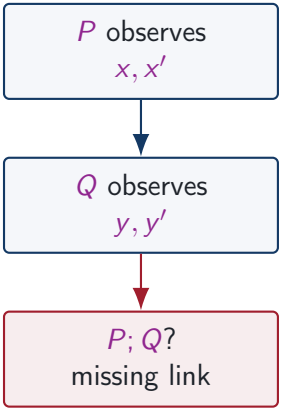
Alphabets are interfaces

Alphabet

$\alpha(P) = \{x, x', y, y'\}$

Pedagogical clarification

- Many semantic errors are interface errors.
- Students cross from syntax to semantics, from relations to designs, from predicates to refinement, or from operational intuitions to denotational models.
- Without noticing that the vocabulary, observations, and ordering have changed.
- Teaching UTP well means making those interfaces visible.



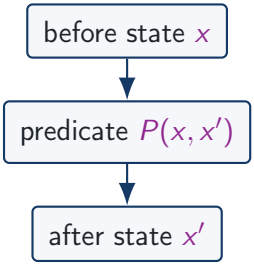
Programs as predicates over observations

Pointwise relations

- Start with the familiar.

  $$x := x + 1 \quad \widehat{=} \quad x' = x + 1$$

- Program: Possible pairs of before and after observations.
- Assignment as a relation is simple.
- But it opens the whole UTP approach.



Teaching payoff

Students learn one relational meaning. Reuse it across all sematics.
Partial correctness unification: specifications, Hoare logic, weakest preconditions.

From relations to assumption-commitment contracts

Refinement calculus

- Designs: the theory of precondition-postcondition contracts.

  $$P \vdash Q \quad \widehat{=} \quad \mathrm{ok} \wedge P \Rightarrow \mathrm{ok}' \wedge Q.$$

- From relations (partial correctness) to designs (total correctness).
- From greatest fixed points to least fixed points.

From partial to total correctness

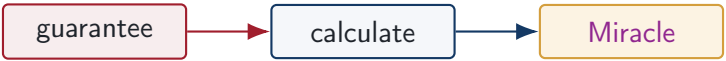
- Stepwise refinement: Morgan's refinement calculus.
- Total correctness unification: specifications, Hoare logic, weakest preconditions.

A micro-example: refinement needs extremes

Complete lattice of programs

- Refinement, lattice order, top (miracle), and bottom (abort).

  Miracle: $x = 0 \vdash x' = 0 \land x' = 1$ abort: $x = 0 \land x = 1 \vdash x' = 0$.
- Important artefacts in the refinement calculus.
- Specifications or intermediate results.



Teaching payoff

The lattice extremes aren't curiosities. They are early diagnostic tools.
The impossible postcondition isn't just wrong.
The calculus tells us exactly how it became wrong.

Refinement as logical strengthening

Understanding refinement

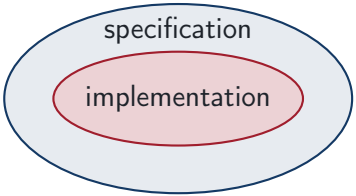
- Make refinement intuitive through informal examples.
- Stronger predicates smaller behaviour sets. Stepwise refinement: make decisions.
- This gives the refinement order its teaching force.
- Formal definition $P \sqsubseteq Q \iff [\, Q \Rightarrow P \,]$. Formalise the informal examples.

Specification

Everything that's acceptable.

Implementation

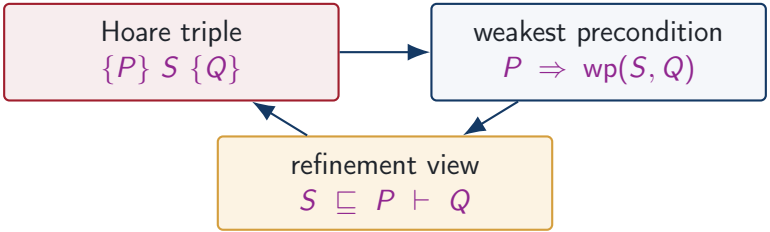
The designer's best implementation



Unifying Hoare logic, weakest preconditions, and refinement

Unification lesson

- Understand three great ideas in software engineering.
- The software engineering trinity: See them as strongly related concepts.



Pedagogical message

These aren't three subjects. They are three presentations of one semantic judgement. It shows that UTP reorganises existing methods rather than replacing them.

Fixed points stop being mysterious

Motivate and educate

- Software engineers and many computer scientists are weak in theory.
- Gently introduce lattices. Make fixed points concrete. Use many examples.
- Fixed points arise because loops and recursion define behaviours recursively.
- This is a gateway to order-theoretic thinking.

$$\text{while } b \text{ do } S \quad = \quad \mu X \bullet ((b \wedge S \,;\, X) \vee (\neg b \wedge \text{Skip}))$$

Programming view

A loop repeats a body until a test fails.
Recursion reduces a parameter.

Semantic view

Solution of a recursive equation.
Which solution?

Invariants become the practical proof method for fixed-point equations.

Compilation as a refinement calculus

Unifying compilation and refinement

- Program compilation is familiar. Make it part of the refinement curriculum.
- Connect formal semantics with compiler correctness, data refinement, optimisation.

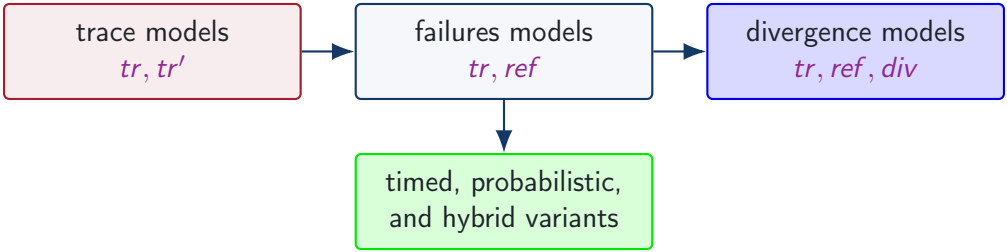
Teaching clarification

Compilation isn't just translation.
It's correctness-preserving movement between semantic theories.

Process algebras as a family of observation theories

Reactive processes

- Establish the relationship between different process algebras.
- UTP compares them by observation structure and healthiness.

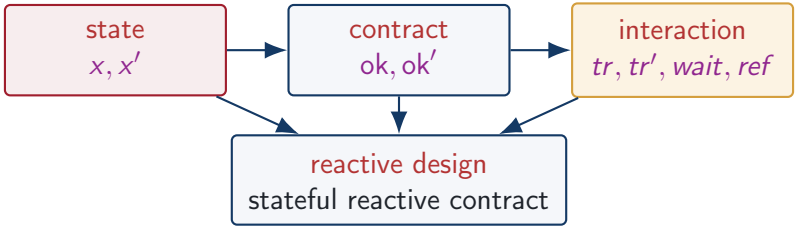


The comparison questions become
What do we observe? What healthiness conditions define valid processes?

Imperative CSP as reactive designs

Diverge from UTP book: CSP contracts

- Imperative CSP = designs (state) + reactive behaviour (concurrency + comms).
- Introduce CSP as the theory of reactive designs: $\mathbf{R} \circ \mathbf{H}$.



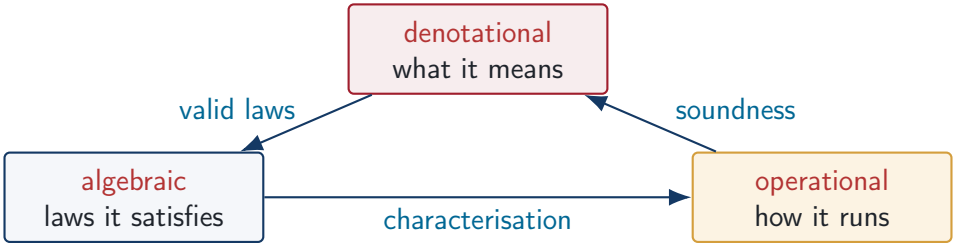
Teaching payoff

Imperative CSP isn't a semantic hybrid accident.
It's a controlled combination of state, contract, and reactivity.

Denotational, algebraic, and operational semantics

Semantic varieties

- Why are there different semantic presentations of languages?



Pedagogical message

They aren't competing semantics. They are mutually justifying views.

Roadmap

The semantic core

From concepts to a course

Feedback, tools, and artefacts

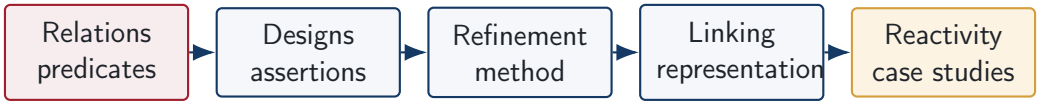
The call to action

A staged course architecture

Pipeline of ideas

- A UTP course has a natural progression.

Sequence scales from semester courses to short schools.



Design principle

Each stage should preserve as much of the previous reasoning discipline as possible.

What students must be able to do

Translate the course architecture into learning outcomes.

Basic skills

- Track alphabets.
- Diagnose interface failures.
- Calculate relational, design meanings.
- Prove and use healthiness closure.

UTP expertise

- Establish refinement steps.
- Handle side conditions.
- Transport specifications.
- Express state contracts.
- Express reactive contracts.

Assessment should test procedural competence, not definition recall.
Take-home assignment and viva voce examination.

Exercise taxonomy: build refinement literacy

Refinement literacy is the key.

Exercise family	What it trains
Alphabet drills	interface discipline
Closure drills	healthiness as calculation
Design algebra	assumption-commitment reasoning
Refinement steps	law-guided development
Linking exercises	semantic transport
Reactive phase change	state to interaction

Teaching rule

Many small calculations beat rare large proofs.

Alphabet drill: diagnose the failure

Teaching alphabets

- This illustrates how to mark explanations, not just final answers.
- It teaches students to see the alphabet as semantic structure.
- What must we check for $P(x, x') \,;\, Q(y, y')$ to be well founded?

Weak answer

"The composition doesn't type-check."

Strong answer

"The after-observation of P isn't connected to the before-observation of Q. Repair by renaming, extending, or inserting a linking relation."

Design algebra drill: find the broken requirement

Concrete assessment style

- Connect refinement, lattice extremes, and requirements diagnosis.
- Example: $P \vdash Q_1 \wedge Q_2$

The exercise isn't only to calculate.
It's to explain which strengthening made the requirement unimplementable.

Refinement step drill: make side conditions visible

Side conditions: where semantics becomes learnable practice.

Refinement chain: $S_0 \sqsubseteq S_1 \sqsubseteq S_2 \sqsubseteq \cdots \sqsubseteq \mathit{Code}$

Every step records the following

Law used, alphabet condition, healthiness condition, monotonicity condition, and definedness condition.

law

alphabet

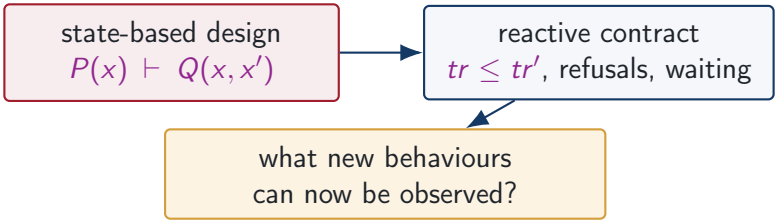
healthiness

monotonicity

definedness

Reactive phase change exercise

Reinforce: observation space expands while proof habits remain stable.



Add the reactive alphabet.
Reactivity is a change of semantic phase.
It's not a new subject.

Roadmap

The semantic core

From concepts to a course

Feedback, tools, and artefacts

The call to action

What feedback students need

Feedback is the most important element of teaching

- Clarifies expectations: what went well, what must be improved.
- Promotes growth mindset: mistakes are learning opportunities, not failures.
- Builds self-regulation: evaluate own work, take ownership, become independent.
- Improves teaching quality: feedback is a two-way street.

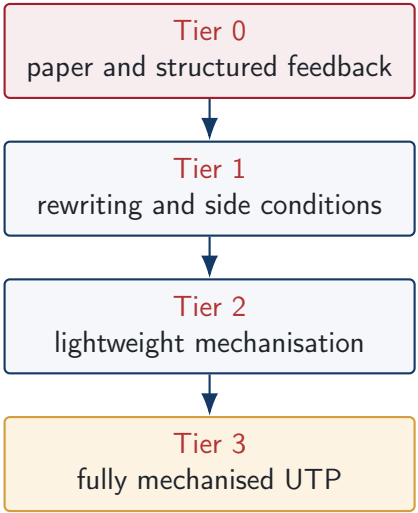
Technical feedback

- Is this law application legal?
- Are side conditions discharged?
- Which alphabet caused failure?
- Is this predicate healthy?
- Does the refinement fail locally?
- Counterexample state or trace?

Tooling principle

Expose the structure of the derivation. Don't hide it behind automation. Use automation to support explanation, not to replace it.

Four tooling tiers

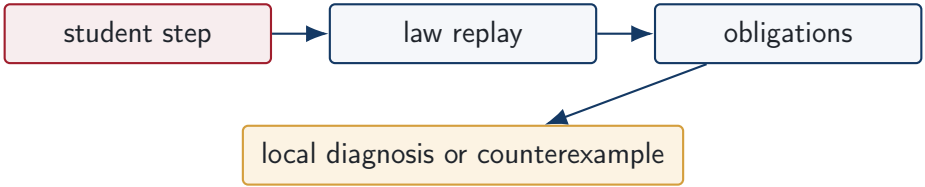


Pragmatic lessons

- In a short course, students may work mainly at Tier 0 or Tier 1: paper-and-pencil calculations, structured rewriting, and explicit side-condition tracking.
- More advanced tiers (lightweight or full mechanisation) can be shown as demonstrations, to indicate that calculations scale and can be checked by tools.
- This prevents tool maximalism: the course is about learning UTP, not about making full tool mastery a prerequisite for understanding.

The most valuable tool output

This is a teaching tool design principle.



- Not just wrong. Show how it's wrong.
- Counterexamples and local diagnoses matter.
- They prevent misconceptions from persisting.

Research-led teaching artefacts

- Produce reusable artefacts. Not just papers, full mechanisations, or polished lecture slides.
- We need compact, shareable units that come directly from research practice.
- Shaped so that students can learn from them.
- Not merely examples. Teaching objects that expose how UTP reasoning works.

Logic, counterexamples, derivations

- Logic-profiled law sheets are collections of UTP laws with assumptions.
- Conditional law holds in classical two-valued logic, fails under strict undefinedness logic, requires totality condition under McCarthy's sequential logic.
- Laws aren't magic; they depend on a semantic profile.
- Counterexample packs are small examples showing where a plausible law fails.
- Students often learn more from a failed law than from a valid one.
- Counterexample shows refinement law fails without a healthiness condition.
- Conditional simplification fails when guards are undefined.
- Replayable derivations are worked calculations students can follow, modify, repeat.
- They might be written on paper, in LaTeX, or in Isabelle/Isar.
- The key feature is that the derivation is not just stated.
- It's structured so that students can replay the reasoning step by step.

Triangles, kernels, benchmarks

- Semantic-triangles connect three semantic presentations.
- Teaching units that connect three presentations of the same idea: denotational semantics, operational semantics, and algebraic laws.
- Students should see semantic definitions, but also execution and calculation.
- Teaching kernels Small mechanised cores: minimal Isabelle developments.
- Compact formal artefacts, containing only the lesson's definitions and laws.
- Not industrial-strength libraries. They are small enough for students to understand.
- Benchmark case studies reusable across courses, tools, and research papers.
- Assignment, conditionals, sequential composition, designs, reactive designs, ...
- Their value is that they provide common reference points.

Assessment, tool configuration

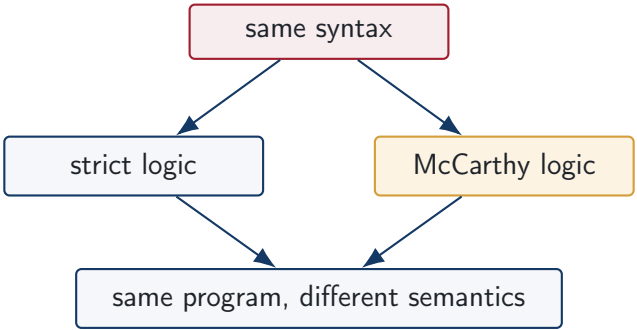
- Assessment rubrics Explicit criteria for judging students' UTP work.
- Have they chosen the right observations?
- Have they stated the healthiness conditions?
- Have they justified each refinement step?
- Have they tracked side conditions?
- Have they explained the semantic assumptions behind a law?
- Reproducible tool configurations Ready-to-run tool setups.
- Isabelle sessions, ROOT files, theory imports, libraries, proof scripts, or Docker/Nix-style environments.
- Reduce accidental difficulty. Students should not spend most of their effort fighting installation details.

Teaching artefacts

- The best artefacts aren't necessarily the largest mechanisations.
- They are the most reusable teaching units.
- The artefacts for teaching UTP aren't necessarily huge mechanised developments.
- They are compact, reusable units that make the semantic structure visible.
- Law sheet: say which logic it assumes.
- Counterexample: show why a tempting law fails.
- Replayable derivation: let students follow the calculation.
- Teaching kernel: expose definitions without overwhelming engineering detail.
- Share these compact artefacts, so research practice becomes teachable material.
- Research-led teaching should package research insight into reusable learning units.
- Not everything has to be a full mechanisation.
- Often the best teaching contribution is a small artefact and one semantic idea.

Semantic variation is teachable

Which logic does UTP use?



Students learn to ask: for which semantic commitments is this law valid?

Evidence: what should we measure?

Measurements

Capability	Evidence
Refinement literacy	multi-step derivations with side conditions
Alphabet competence	accurate diagnosis of interface failures
Healthiness reasoning	closure proofs and simplification
Transfer	reuse of laws across theory families
Tool learning	improved corrections after feedback

FMTea opportunity

Small cohorts can still report useful evidence: misconceptions, error classes, and feedback that fixes them.

Roadmap

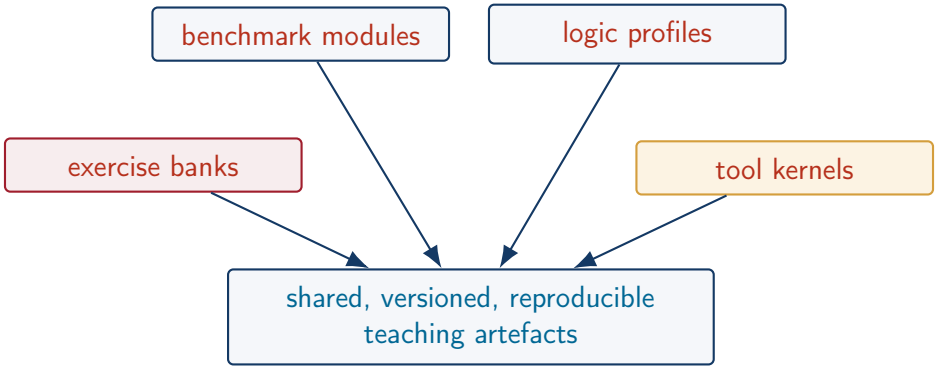
The semantic core

From concepts to a course

Feedback, tools, and artefacts

The call to action

A community repository for UTP teaching



Make the method more visible, repeatable, and shareable.

Six teaching maxims

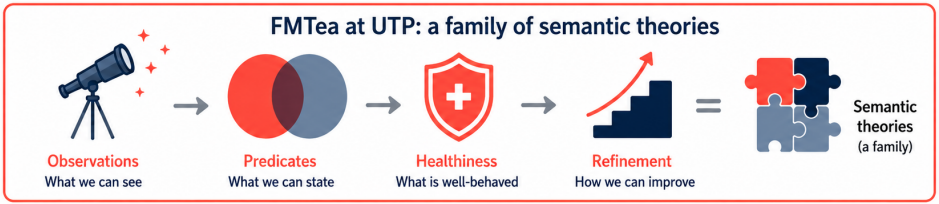
UTP is difficult to teach

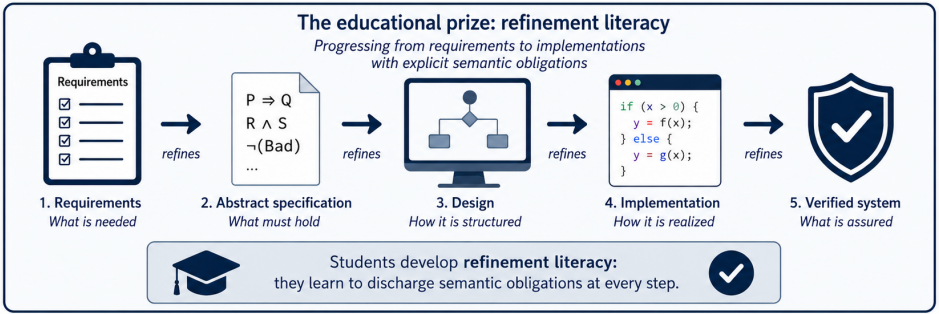
- It's not because of the mathematics, but because movement is often implicit.
- We move between syntax, semantics, algebra, refinement, proof, tool support.
- Teaching task: Slow that movement down, name it, make it inspectable.
- Don't make UTP less theoretical; make the route through the theory more visible.

How to teach UTP

1. Teach procedures, not just concepts.
2. Treat alphabets as interfaces.
3. Make side conditions visible.
4. Use miracles and chaos diagnostically.
5. Connect Hoare logic, wp, and refinement early.
6. Relate semantic presentations by explicit proof obligations.

The take-home story





Questions for FMTea

FMTea Community initiative

1. Which UTP teaching kernels should we build first?
2. Which misconceptions should we benchmark?
3. How do we make semantic variation accessible without hiding the mathematics?

Thank you!