# Isabelle/UTP: A mechanised theory engineering framework

Simon Foster  Frank Zeyda  Jim Woodcock

University of York

June 3, 2014

# Outline

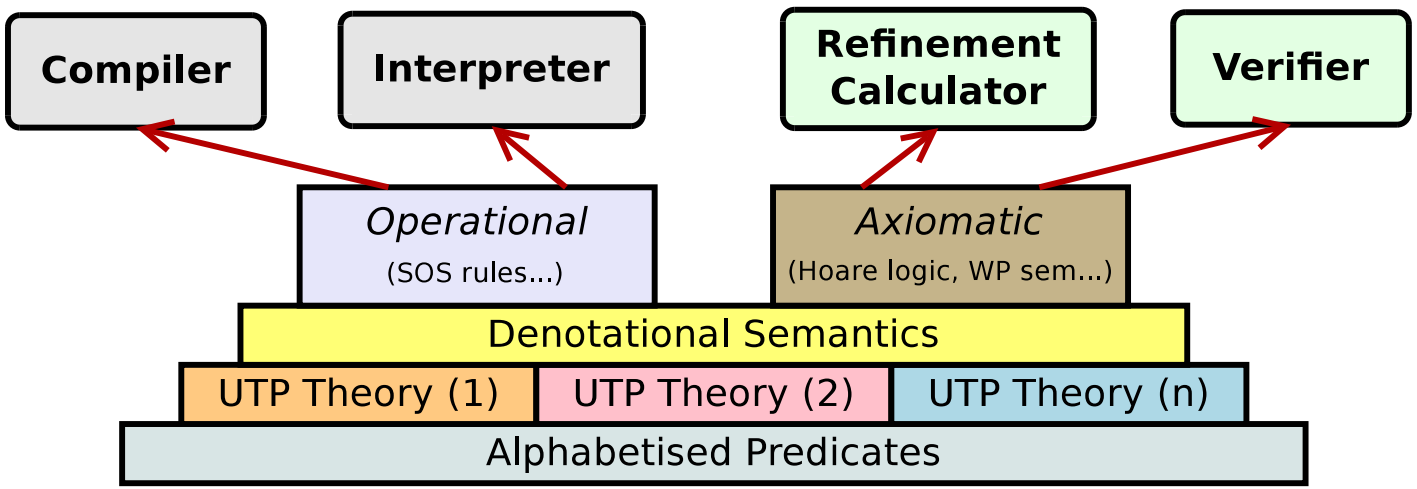
# Outline

# Semantic Heterogeneity

- languages increasingly semantically heterogeneous
- need to compose features from a number of areas
  - concurrency, object-orientation, design by contract
  - continuous time, probability, $\cdots$
- case in point: Cyber-Physical Systems
- our solution: Unifying Theories of Programming
- how to put this to work? (cf. UToPiA)
- need dependable tools

# Tool-chain foundations

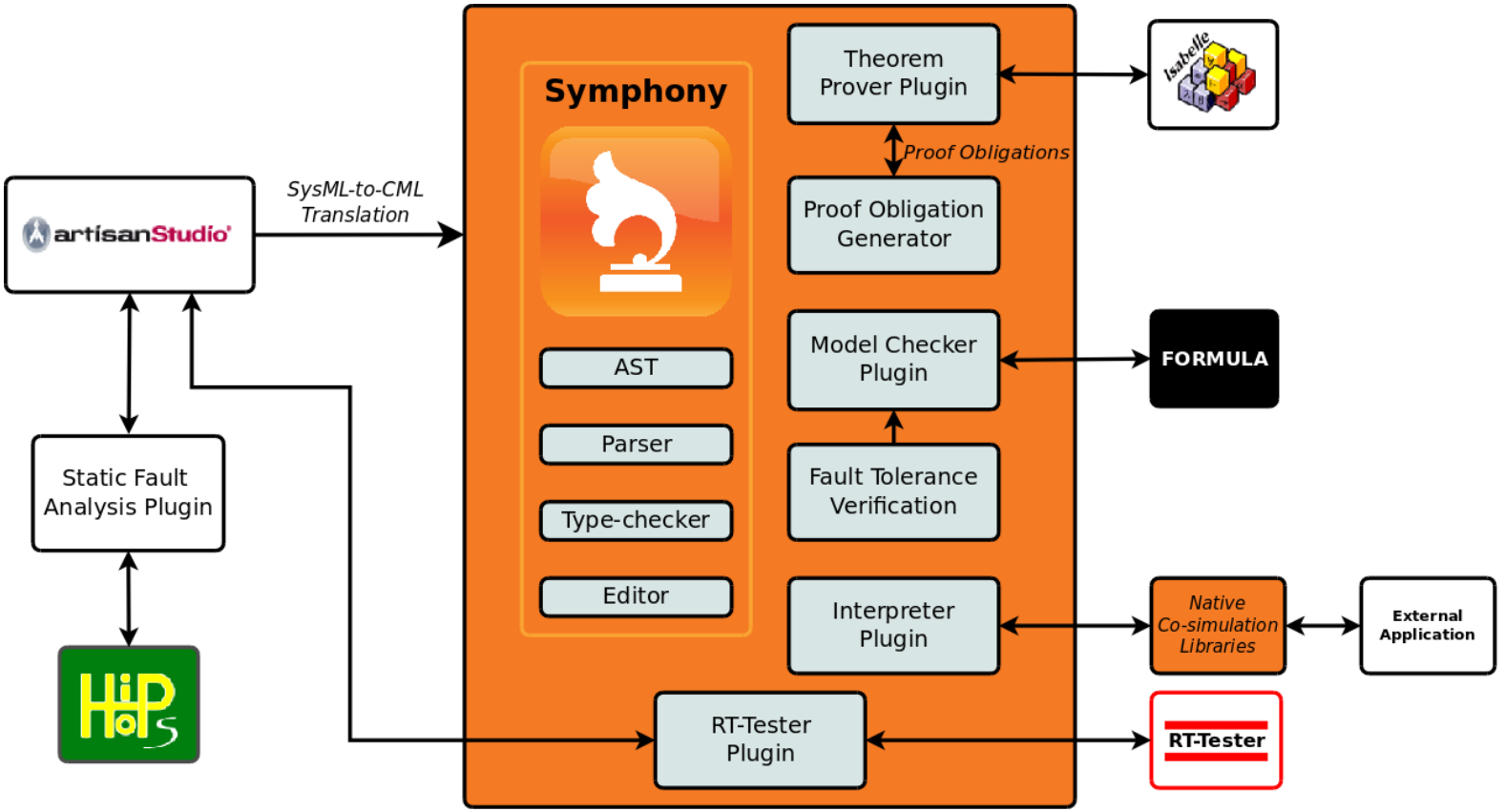
- a UTP-based tool-chain requires firm mechanised foundations
- mechanically construct and verify UTP theories
- formally prove correspondence between semantic models



# Motivation: Symphony

- developed on the COMPASS project
- Eclipse-based modelling environment for Systems of Systems
- CML – a formal modelling language based on VDM and Circus
- denotational and operational semantics based in the UTP
- variety of components for analysis of CML models
- unified semantics ensure integrity of analysis results
- download: http://www.symphonytool.org

# Symphony Components



# Mechanisation Criteria

1. **Consistency**: is it possible to prove contradictions?
2. **Expressivity**: can we express all the laws need in UTP?
3. **Proof Automation**: how do we make use of existing tools?
4. **Well-formedness**: how to ensure predicates are well-formed?
5. **Modularity**: how to ensure separation of concerns?

# Current mechanisations

- ProofPower-Z UTP – "deep" embedding in ProofPower
  - expressive predicate model
  - little proof automation
  - manual type checking
- Isabelle/Circus – "shallow" embedding in Isabelle/HOL
  - directly use type system
  - directly use tactics
  - no explicit support for variables
- U·(TP)$^2$ – fresh UTP theorem prover in Haskell
  - faithful implementation of the logic
  - consistency?
  - cannot take advantage of automated proof

# Isabelle/UTP

- a semantic embedding of the UTP logic in Isabelle/HOL
- inspiration: ProofPower-Z UTP and Isabelle/Circus
- aim: reconcile (as much as possible) deep and shallow
- why?
  - deep embeddings best for meta-logical proofs (more control)
  - allows embeddings best for verification (efficiency)
  - can we do both in the same system?
- how? harness the HOL type-system, proof tactics and laws
- but still retain deep concepts from ProofPower-Z UTP
- enable a more disciplined approach to UTP theory engineering

# Why Isabelle?

- Higher Order Logic – ideal for semantic embeddings
- LCF architecture – reliability of proofs
- (dis)proof automation – simp, auto, sledgehammer, nitpick...
- readable proofs – supported by Isar

- **overall:** balance of mathematical principal and automation
- **aim:** achieve a faithful embedding whilst harnessing Isabelle

# Outline

# Overview

- purely semantic model of predicates (no fixed syntax)
- generic value space model supporting different languages
- layered predicate model consisting of
  - core predicates (binding sets – cf. Z in HOL)
  - alphabetised predicates (core predicate + alphabet)
- expression model supporting substitutions $P[e/x]$
- usual UTP operators (predicates, relations, etc.)
- based on ProofPower-Z UTP but well-formed by construction

# Value space axiomatisation

- type-classes: local theory context with a signature and assumptions, linked to a polymorphic type variable

```
class DEFINED =
  fixes Defined  :: 'a ⇒ bool (𝒟)

class VALUE = DEFINED +
  fixes utype-rel :: 'a ⇒ nat ⇒ bool (infix :ᵤ 50)
  assumes utype-nonempty: ∃ t v. v :ᵤ t ∧ 𝒟 v

typedef 'a utype = {t. ∃ v::'a. v :ᵤ t ∧ 𝒟 v}
  by (smt mem-Collect-eq utype-nonempty)

definition type-rel :: 'a ⇒ 'a utype ⇒ bool (infix : 50) where
  x : t ⟷ x :ᵤ Rep-utype t
```

# Bindings and Core Predicates

- variables: name + type + auxiliary flag
- bindings: (total) functions from variables to values
- core predicate: set of bindings

**typedef** *'a binding* = $\{b\ .\ \forall\ v{::}'a\ uvar.\ b\ v \rhd v\}$

**typedef** *'a pred* = *UNIV* :: *'a binding set set*
**morphisms** *rep-pred abs-pred* **..**

**notation** *rep-pred* $(\langle\text{-}\rangle_p)$

- binding override: $b_1 \oplus_b b_2$ **on** *vs*
- binding update: $b(x :=_b v)$

# Predicate Operators

**lift-definition** *TrueP* :: *'a pred* **is** *UNIV* :: *'a binding set* **.**

**lift-definition** *NotP* :: *'a pred* ⇒ *'a pred* **is** *uminus* **.**

**lift-definition** *AndP* :: *'a pred* ⇒ *'a pred* ⇒ *'a pred*
**is** $\lambda\ x\ y.\ (x \cap y :: \text{'a binding set})$ **.**

**lift-definition** *AndDistP* :: *'a pred set* ⇒ *'a pred*
**is** $\lambda\ ps.\ \bigcap\ \{\langle p \rangle_p \mid p.\ p \in ps\}$ **.**

**lift-definition** *ExistsP* :: *'a uvar set* ⇒ *'a pred* ⇒ *'a pred*
**is** $\lambda\ vs\ p.\ \{b_1 \oplus_b b_2\ on\ vs \mid b_1\ b_2.\ b_1 \in p\}$ **.**

# Expressions

**typedef** $'a\ expr = \{f :: 'a\ binding \Rightarrow 'a.\ \exists\ t.\ \forall\ b.\ f\ b : t\}$

**lift-definition** $LitE :: 'a\ utype \Rightarrow 'a \Rightarrow 'a\ expr$
  **is** $\lambda\ t\ v\ \ b :: 'a\ binding.\ if\ (v : t)\ then\ v\ else\ somev(t)$
  **by** $(metis\ (full\text{-}types)\ somev\text{-}type)$

**lift-definition** $EqualP :: 'a\ expr \Rightarrow 'a\ expr \Rightarrow 'a\ pred$
  **is** $\lambda\ u\ v.\ \{b :: 'a\ binding.\ u(b) =\ \ v(b)\}$ **..**

**lift-definition** $SubstP :: 'a\ pred \Rightarrow 'a\ expr \Rightarrow 'a\ uvar \Rightarrow 'a\ pred$
  **is** $\lambda\ p\ v\ x.\ \{b.\ b(x :=_b v(b)) \in p\}$ **..**

# Alphabets and Unrestriction

- $xs \sharp P$ – $P$'s value independent of variables $xs$

```
definition UNR :: 'a uvar set ⇒ 'a pred ⇒ bool (infixr ♯ 60) where
  UNR vs p = (∀ b₁ ∈ ⟨p⟩ₚ . ∀ b₂. b₁ ⊕ᵦ b₂ on vs ∈ ⟨p⟩ₚ)

lemma UNR-TrueP: vs ♯ true
  by (metis TrueP.rep-eq UNIV-I UNR-def)

lemma UNR-AndP: ⟦ vs ♯ p; vs ♯ q ⟧ ⟹ vs ♯ p ∧ₚ q
  by (smt AndP.rep-eq IntD1 IntD2 IntI UNR-def)

typedef 'a apred = {(a :: 'a uvar set , p :: 'a pred). − a ♯ p}
  by (metis UNR-TrueP mem-Collect-eq split-conv)

lift-definition AndA :: 'a apred ⇒ 'a apred ⇒ 'a apred
  is λ (A₁, P) (A₂, Q). (A₁ ∪ A₂, P ∧ₚ Q)
```

# Why the dichotomy?

- **core predicates**
  - provide an easy link into existing HOL algberas
  - (e.g. a single relational identity)
  - easier syntax – no need for alphabet in $x := v$
  - proofs need not respect the alphabet in each step
- **alphabetised predicates**
  - are necessary for UTP (particular theories)
  - lattice / fixed-point theory don't quite work for core preds
  - restricted to a finite set of variables
  - can be easier to prove freeness properties
- can often switch between the two in proof

# Outline

# Typing

- expression model is typed, but via an Isabelle predicate
- hence type correctness must be manually proved
- unacceptable for predicates of reasonable size
- problem of a deep embedding: we have captured typing
- cf. Isabelle/Circus which directly uses the type system
- but there are advantages to deep e.g.
  - enables implementation of dependent products
  - can express sets of heterogeneous variables (alphabets)
- what's the solution?

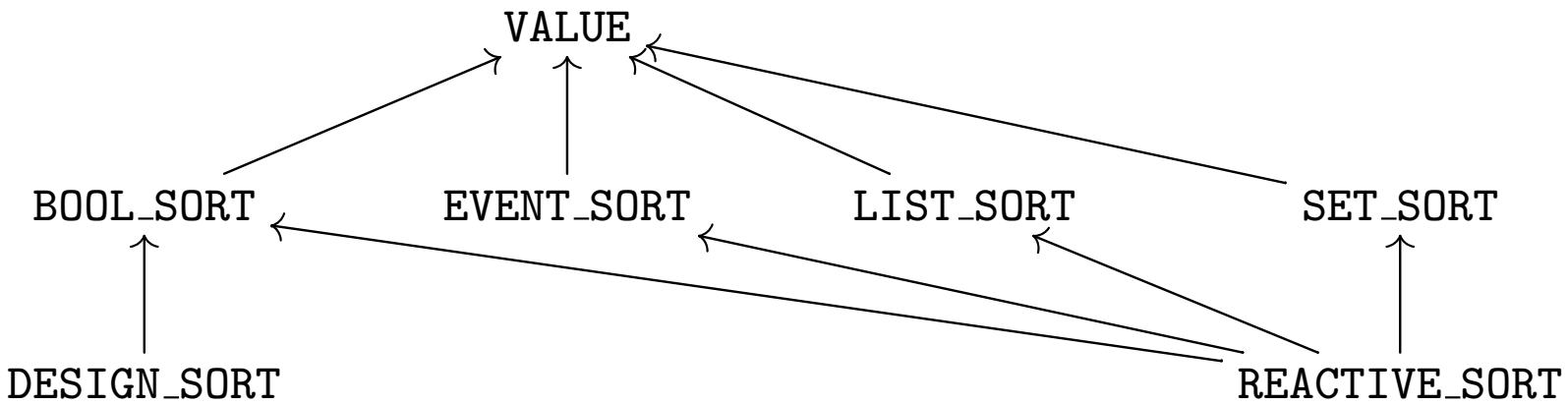
# Part 1: Sort Classes

- a sort class extends VALUE with a type axiomatisation

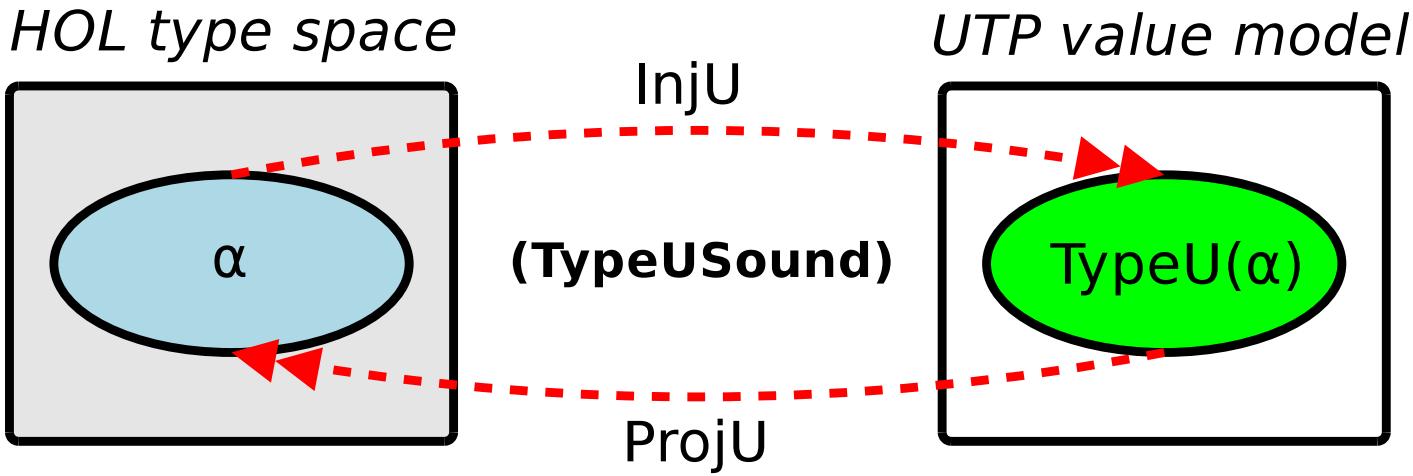
```
class BOOL-SORT = VALUE +
  fixes MkBool :: bool ⇒ 'a
  fixes DestBool :: 'a ⇒ bool
  fixes BoolType  :: 'a utype
  assumes Inverse [simp] : DestBool (MkBool b) = b
  and     BoolType-dcarrier: dcarrier BoolType = range MkBool
```

- InjU and ProjU represented by polymorphic constants
- use sort classes to populate particular instances

# Part 1: Sort Classes



# Part 2: Type embeddings




- sort classes can provide instances for specific types

# Typed Expressions

```
typedef ('a, 'm) pvar = UNIV :: (NAME * bool) set
  by auto

typedef ('a, 'm) pexpr = UNIV :: ('m binding ⇒ 'a) set
  morphisms DestPExpr MkPExpr ..

lift-definition LitPE :: 'a ⇒ ('a, 'm) pexpr
  is λ v. λ b :: 'm binding ⇒ v .
```

# Type erasure

- HOL disallows heterogeneous collections of data
- provides the ability to move from HOL typed to UTP typed
- operator written: $x\downarrow$

```
definition erase-pvar :: ('a, 'm :: VALUE) pvar ⇒ 'm uvar where
erase-pvar x = MkVar (pvname x) TYPEU('a) (pvaux x)

definition erase-pexpr :: ('a, 'm :: VALUE) pexpr ⇒ 'm uexpr where
erase-pexpr e = MkExpr (λ b. InjU (DestPExpr e b))
```

# Outline

# Proof in UTP

## Question

- how to make use of Isabelle proof tools in the UTP?
- want to reach the same level of complexity as book proofs
- solution: transfer Isabelle proofs to UTP proofs

# Proof Strategy – Tactics

## Proof by Interpretation

- identify subtheories within UTP
- use HOL proof procedures to discharge goals
- fundamental idea in Isabelle

- tactics developed so far:
  - utp-pred-tac: UTP predicates as HOL predicates
    - handles std. logic operators $\wedge$, $\vee$, $\Rightarrow$, $\sqsubseteq$, $\neg$, $\exists$, $\forall$
  - utp-rel-tac: UTP predicates as HOL relations
    - classical binary relations: $\mathbb{P}\,(A \times A)$
    - handles most rel operators $\cup$, $\cap$, $;$, $\mathrm{II}$, false, $^\smile$
  - utp-xrel-tac: well-formed UTP relations as HOL relations
    - adds $\neg$ and true: complete relation algebra

# What's in a UTP tactic?

1. interpretation function – from the UTP to the target domain
2. transfer rules – transfer results from target into UTP domain
3. congruence rules – map operators of UTP to the target

# Relation Tactic: utp-rel-tac

## Interpretation Function

$$\text{EvalR} :: {'a}\ \text{pred} \Rightarrow ({'a}\ \text{rel\_binding})\ \text{rel}\ (\llbracket \_ \rrbracket \mathcal{R})$$

## Transfer Theorems

$$(P = Q) \longleftrightarrow (\llbracket P \rrbracket \mathcal{R} = \llbracket Q \rrbracket \mathcal{R})$$
$$(P \sqsubseteq Q) \longleftrightarrow (\llbracket P \rrbracket \mathcal{R} \supseteq \llbracket Q \rrbracket \mathcal{R})$$

## Congruence Rules (Selection)

$$\begin{aligned}
\llbracket \text{false} \rrbracket \mathcal{R} &= \emptyset \\
\llbracket P \wedge_p Q \rrbracket \mathcal{R} &= \llbracket P \rrbracket \mathcal{R} \cap \llbracket Q \rrbracket \mathcal{R} \\
\llbracket P \vee_p Q \rrbracket \mathcal{R} &= \llbracket P \rrbracket \mathcal{R} \cup \llbracket Q \rrbracket \mathcal{R} \\
\llbracket \text{II} \rrbracket \mathcal{R} &= \text{Id} \\
\llbracket P \fatsemi Q \rrbracket \mathcal{R} &= \llbracket P \rrbracket \mathcal{R} \circ \llbracket Q \rrbracket \mathcal{R}
\end{aligned}$$

# Algebraic Laws Selection

| | |
|---|---|
| SemiR-SkipR-left | $: \mathrm{II} \fatsemi P = P$ |
| SemiR-assoc | $: P \fatsemi (Q \fatsemi R) = (P \fatsemi Q) \fatsemi R$ |
| SemiR-AndP-right-pre | $: P \fatsemi (c \wedge Q) = (P \wedge c') \fatsemi Q$ |
| SemiR-CondR-distr | $: (P \lhd b \rhd Q) \fatsemi R = (P \fatsemi R) \lhd b \rhd (Q \fatsemi R)$ |
| SemiR-extract-var | $: P \fatsemi Q = (\exists x''.P[x''/x']; Q[x''\|x])$ |
| AssignR-SemiR-left | $: (x := e \fatsemi p) = p[e/x]$ |
| AssertR-SemiR | $: b_\perp \fatsemi c_\perp = (b \wedge c)_\perp$ |
| IterP-false | $:$ **while** *false* **do** $P$ **od** $= \mathrm{II}$ |

# Outline

# Hoare Logic in UTP

- Hoare triple: $\{p\}Q\{r\}$
- $p$ and $r$ are assertions and $Q$ is a command
- partial correctness

$$\{p\}Q\{r\} \quad \triangleq \quad (p \Rightarrow r') \sqsubseteq Q$$

- have proved the standard laws in Isabelle/UTP
- associated tactics can be used for verifying simple programs

# Outline

# Conclusion

- reasonably faithful mechanisation of UTP
- access to most proof facilities of Isabelle/HOL
- fully extensible at tactic, parser, and support law levels
- standard laws for predicates and relations
- theories of designs and reactive processes
- theorem prover for CML in Symphony
- **future:** tactics, theories, calculi, law library, total correctness

- Website: http://www.cs.york.ac.uk/~simonf/utp-isabelle/
- COMPASS: http://www.compass-research.eu