Isabelle/UTP
A Verification Toolbox for Unifying Theories

Simon Foster  Jim Woodcock  Kangfeng Ye
{firstname.lastname}@york.ac.uk

University of York, UK

Friday 22\textsuperscript{nd} September, 2017

INTO-CPS

http://into-cps.au.dk/
VeTSS Project

“Mechanised Assume-Guarantee Reasoning for Control Law Diagrams via Circus”

- AG proof support for discrete time Simulink diagrams
- Circus: stateful reactive language extending CSP
- use of reactive contracts to specify properties
- develop a library of examples and two case studies
- mechanised proof support for Simulink in Isabelle/UTP
- researcher: Dr. Kangfeng Ye (Randall)
Unifying Theories of Programming

- formal semantics framework from Tony Hoare and He Jifeng
Unifying Theories of Programming

- formal semantics framework from Tony Hoare and He Jifeng
- drives to find theories that unify computational paradigms
  - imperative and functional programming
  - sequential and concurrent computation
  - data structures and object orientation
  - real-time and hybrid systems

Use alphabetised relational calculus as a lingua franca
- programs-as-predicates: specification + implementation
- link different semantic models (operational, axiomatic etc.)
- build verification tools for various paradigms
Unifying Theories of Programming

- formal semantics framework from Tony Hoare and He Jifeng
- drives to find theories that unify computational paradigms
  - imperative and functional programming
  - sequential and concurrent computation
  - data structures and object orientation
  - real-time and hybrid systems

Can we find fundamental laws that characterise their commonalities and highlight their differences?
Unifying Theories of Programming

- formal semantics framework from Tony Hoare and He Jifeng
- drives to find theories that unify computational paradigms
  - imperative and functional programming
  - sequential and concurrent computation
  - data structures and object orientation
  - real-time and hybrid systems

Can we find fundamental laws that characterise their commonalities and highlight their differences?

- use alphabetised relational calculus as a lingua franca
- programs-as-predicates: specification + implementation
- link different semantic models (operational, axiomatic etc.)
- build verification tools for various paradigms
Example: Operational Semantics and Hoare Calculus

**Definition (Transition Relation)**

\[(\sigma_1, P_1) \rightarrow (\sigma_2, P_2) \triangleq \langle \sigma_1 \rangle ; P_1 \sqsubseteq \langle \sigma_2 \rangle ; P_2\]

**Theorem (Operational Laws)**

\[
\begin{align*}
(\sigma, P) \rightarrow (\rho, Q) & \quad \text{SEQ-STEP} \\
(\sigma, P ; R) \rightarrow (\rho, Q ; R) & \\
\sigma \models c & \quad \text{COND-TRUE} \\
(\sigma, \text{if } c \text{ then } P \text{ else } Q) \rightarrow (\sigma, P) & \\
\sigma, x := v \rightarrow (\sigma(x := \sigma \uparrow v), \top) & \quad \text{ASSIGN} \\
\sigma \models c & \quad \text{ITER-COPY} \\
(\sigma, \text{while } c \text{ do } P) \rightarrow (\sigma, P ; \text{while } c \text{ do } P) &
\end{align*}
\]
## Example: Operational Semantics and Hoare Calculus

### Definition (Transition Relation)

\[(\sigma_1, P_1) \rightarrow (\sigma_2, P_2) \triangleq \langle \sigma_1 \rangle ; P_1 \sqsubseteq \langle \sigma_2 \rangle ; P_2\]

### Definition (Hoare Calculus)

\[
\{ p \} \quad Q \quad \{ r \} \triangleq (p \Rightarrow r') \sqsubseteq Q
\]
Example: Operational Semantics and Hoare Calculus

Definition (Transition Relation)

\[(σ_1, P_1) \rightarrow (σ_2, P_2) \triangleq (σ_1) ; P_1 \sqsubseteq (σ_2) ; P_2\]

Definition (Hoare Calculus)

\[\{ p \} Q \{ r \} \triangleq (p \Rightarrow r') \sqsubseteq Q\]

Theorem (Linking)

\[\{ p \} Q \{ r \} \iff \sigma_1 \models p \quad (σ_1, Q) \rightarrow (σ_2, \emptyset) \quad \sigma_2 \models r\]

- operators are denotations, laws are theorems
- we apply this technique to more complex computational paradigms, such as concurrent and hybrid systems
INtegrated TOolchain for Cyber-Physical Systems

INTO-CPS Multi-Modelling

\[
\dot{\psi} = \frac{\omega_r - \omega_i}{T_e}
\]
\[
\begin{bmatrix}
\dot{x} \\
\dot{y}
\end{bmatrix} = V_c \begin{bmatrix}
\cos(\psi) \\
\sin(\psi)
\end{bmatrix}
\]

UTP Multi-Model Semantics

```c
#include <stdio.h>
define MAX 10
int main()
{
    char array[MAX], c = 0;
    int d = 1, k = 0, i, j;
    do
        scanf("%c", &array[k]);
        while (array[k] != 0);
    for (int i = 0; i < MAX; i++)
    {
        float pf;
        int mx = (int)(array[0][i] + 0.5);
        pf = (float)mx;
        printf("%3.2f\n", pf);
    }
}
```

```
class Controller
{
    values
    setpoint : real = 20;
    tolerance : real = 0.5;

    upperLimit : real = setPoint + tolerance;
    lowerLimit : real = setPoint - tolerance;

    instance variables
    dampenActuator : Actuator;
    valueActuator : Actuator;
    temperature : scalar;
    faultStatus : real = 0.13;
    validStatus : real = 0;
}
```

Zone Temperature

- Area 1 Temperature
- Area 2 Temperature
Vision: UTP CPS Verification Foundations

- **Object Orientation**
- **Differential Equations**
- **Concurrency**
- **Hybrid Systems**
- **State**
- **Real-time**
- **Reactive Systems**

**Unifying Theories of Programming**

- **Graphical Notations**
- **Contracts**

**Isabelle/HOL**
Isabelle/UTP

- a verification toolbox for the UTP based on Isabelle/HOL
- relational calculus, proof tactics, and algebraic laws
Isabelle/UTP

- a verification toolbox for the UTP based on Isabelle/HOL
- relational calculus, proof tactics, and algebraic laws
- define syntax for programs and create verification calculi
- via formalisation of semantic “building blocks” (UTP theories)
- utilise Isabelle’s powerful proof automation for verification
Isabelle/UTP

- a verification toolbox for the UTP based on Isabelle/HOL
- relational calculus, proof tactics, and algebraic laws
- define syntax for programs and create verification calculi
- via formalisation of semantic “building blocks” (UTP theories)
- utilise Isabelle’s powerful proof automation for verification
- formalise links between domains using Galois connection
- large library of formalised algebraic laws of programming
**Examples**

**lemma hoare_ex_1:**
```
"\{true\}(z := &x) ◦ (\&x ≥_u \&y) ▽ (z := \&y) \{&z =_u \max_u (\&x, \&y)\}_u"
```

by (hoare_auto)

**lemma hoare_ex_2:**
```
assumes "X > 0" "Y > 0"
shows
"\{&x =_u «X» ∧ \&y =_u «Y»\}
while ¬(\&x =_u \&y)
  invr \&x >_u 0 ∧ \&y >_u 0 ∧ (gcd_u(\&x, \&y) =_u gcd_u(«X», «Y»))
  do
    (x := (\&x - \&y)) ◦ (\&x >_u \&y) ▽ (y := (\&y - \&x))
  od
\{&x =_u gcd_u(«X», «Y»)\}_u"
```

using assms by (hoare_auto, (metis gcd.commute gcd_diff1)+)
Examples

definition Pay :: "index ⇒ index ⇒ money ⇒ action_mdx" where
Pay i j n =
  pay.(("i", "j", "n") u) →
  ((reject.("i") → Skip)
   ? "i" = u "j" ∨ "i" ∉ u dom u(&accts) ∨ "n" ≤ u 0 ∨ "n" > u &accts("i") a > u
   {accts["i"]} :=C (&accts("i") a - "n") ;
   {accts["j"]} :=C (&accts("j") a + "n") ;
   accept.("i") → Skip)"

definition PaySet :: "index ⇒ (index × index × money) set" where
[upred_defs]: "PaySet cardNum = {(i, j, k). i < cardNum ∧ j < cardNum ∧ i ≠ j}"

definition AllPay :: "index ⇒ action_mdx" where
"AllPay cardNum = (Π (i, j, n) ∈ PaySet cardNum • Pay i j n)"
Examples

```ml
theorem money_constant:
  assumes "finite cards" "i ∈ cards" "j ∈ cards" "i ≠ j"
  shows "[dom_u(&accts) = u "cards" \vdash true | sum_u($accts) = u sum_u($accts')] \subseteq Pay i j n"
  -- {
  theorem money_constant:
  assumes "finite cards" "i ∈ cards" "j ∈ cards" "i ≠ j"
  shows "[dom_u(&accts) = u "cards" \vdash true | sum_u($accts) = u sum_u($accts')] \subseteq Pay i j n"
  -- {* We first calculate the reactive design contract and apply refinement introduction *}
  proof (simp add: assms Pay_contract, rule CRD_refine_rdes)

  -- {* Three proof obligations result for the pre/periph/postconditions. The first requires us to
  show that the contract's precondition is weakened by the implementation precondition.
  It is because the implementation's precondition is under the assumption of receiving an
  input and the money amount constraints. We discharge by first calculating the precondition,
  as done above, and then using the relational calculus tactic. *}

  from assms
  show "\[dom_u(&accts) = u "cards"]_S< =>
    I(true,(pay("i", "j", "n")_u)) =>
    ![i \in dom_u(&accts) \land n \leq u 0 \lor \&accts("i")_u < u "n"] \lor
    ![i \in dom_u(&accts) \land j \in dom_u(&accts)]_S<" by (rel_auto)
```
Examples

\textbf{theorem} extChoice\_commute:
  \textbf{assumes} "\textit{P is NCSP}" "\textit{Q is NCSP}"
  \textbf{shows} "\textit{P} □ \textit{Q} = \textit{Q} □ \textit{P}"
  \textbf{by} (rdes\_eq \textit{cls}: \textit{assms})

\textbf{theorem} extChoice\_assign:
  \textbf{assumes} "\textit{P is NCSP}" "\textit{Q is NCSP}"
  \textbf{shows} "\textit{x := v}; (\textit{P} □ \textit{Q}) = (\textit{x := v}; \textit{P}) □ (\textit{x := v}; \textit{Q})"
  \textbf{by} (rdes\_eq \textit{cls}: \textit{assms})

\textbf{theorem} stop\_seq:
  \textbf{assumes} "\textit{P is NCSP}"
  \textbf{shows} "\textit{Stop}; \textit{P} = \textit{Stop}"
  \textbf{by} (rdes\_eq \textit{cls}: \textit{assms})
**Integrated Toolchain for Cyber-Physical Systems**

Examples

```plaintext
definition
  "BrakingTrain =
    c:accel, c:vel, c:pos := «normal_deceleration», «max_speed», «0» ;
    {&accel,&vel,&pos} • «train_ode»_h until_h ($vel' ≤_u 0) ;
    c:accel := 0"

theorem braking_train_pos_le:
  "($c:accel' =_u 0 \land |$pos' ≤_h 44|_h) ⊑ BrakingTrain" (is "?lhs ⊑ ?rhs")

proof -
  -- {* Solve ODE, replacing it with an explicit solution: @{term train_sol}. *}
  have "?rhs =
    c:accel, c:vel, c:pos := «-1.4», «4.16», «0» ;
    {&accel,&vel,&pos} ←_h «train_sol»(&accel,&vel,&pos)_{«time»}_a until_h ($vel' ≤_u 0) ;
    c:accel := 0"
  by (simp only: BrakingTrain_def train_sol)

  -- {* Set up initial values for the ODE solution using assigned variables. *}
  also have "... =
    {&accel,&vel,&pos} ←_h «train_sol(-1.4,4.16,0)(time)» until_h ($vel' ≤_u 0) ;
    c:accel := 0"
  by (simp add: assigns_r_comp usubst unrest alpha, literalise, simp)
```
Conclusion

- UTP enables a holistic approach to formal semantics
- Isabelle/UTP: computational theories $\rightarrow$ verification tools
- wide spectrum of paradigms supported
Conclusion

- UTP enables a holistic approach to formal semantics
- Isabelle/UTP: computational theories → verification tools
- wide spectrum of paradigms supported
- still much more work to be done
- more UTP theories to mechanise (objects, real-time, etc.)
- performance and scalability
- VeTSS: reasoning about discrete-time Simulink diagrams
Conclusion

- UTP enables a holistic approach to formal semantics
- Isabelle/UTP: computational theories → verification tools
- wide spectrum of paradigms supported
- still much more work to be done
- more UTP theories to mechanise (objects, real-time, etc.)
- performance and scalability
- VeTSS: reasoning about discrete-time Simulink diagrams

Isabelle/UTP: http://www.cs.york.ac.uk/~simonf/utp-isabelle
GitHub: https://github.com/isabelle-utp/utp-main
Email: simon.foster@york.ac.uk
References


- Isabelle/UTP. [https://github.com/isabelle-utp/utp-main](https://github.com/isabelle-utp/utp-main)