Foundations for Simulink diagrams in UTP
Unifying Theories of Hybrid and Reactive Programming

Simon Foster  Jeremy Jacob
Jim Woodcock  Frank Zeyda

University of York, UK
(with thanks to University of Teesside, UK)

Tuesday 30th May, 2017

INTO-CPS

http://into-cps.au.dk/
Outline

Background

Reactive Systems

Unifying Reactive and Hybrid

Towards Simulink Block Semantics

Conclusion
Outline

Background

Reactive Systems

Unifying Reactive and Hybrid

Towards Simulink Block Semantics

Conclusion
Unifying Theories of Programming

- modelling languages have heterogeneous semantics
- UTP: framework for definition and study of formal semantics
- based on “programs-as-predicates”; enables unification
- look at different theoretical aspects in isolation
- formalise aspects as UTP theories
  - alphabet – observational variables
  - signature – operators of the language
  - healthiness conditions – define the theory’s domain
- build the theory supermarket – enable reuse
- study relationships between different modelling languages
- unify different semantic presentations
  - denotational, operational, algebraic, axiomatic...
Programs as Predicates

- observable behaviour encoded in first-order predicate calculus
- relational predicates over input, output variables \((x / x')\)
- alphabet gives the domain of possible observations
- we thus denote the core programming operators

\[
x := v \triangleq x' = v \land y' = y
\]

\[
P ; Q \triangleq \exists x_0 \cdot P[x_0/x'] \land Q[x_0/x]
\]

\[
P \sqcap Q \triangleq P \lor Q
\]

\[
P \bowtie b \bowtie Q \triangleq (b \land P) \lor (\neg b \land Q)
\]

\[
P^{\omega} \triangleq \mu X \cdot P ; X
\]

- definitions support proof of algebraic laws of programming
- natural notion of refinement: \(P \sqsubseteq Q \iff P = P \sqcap Q\)
UTP theory example

- simple language for real-time programs
- **alphabet**: what is observable in particular model
  - clock: $time, time' : \mathbb{N}$
  - program state: $st, st' : \Sigma$
- **healthiness conditions**: $time \leq time'$
  - encoded as idempotent and montonic functions
  - $HT(P) = P \land time \leq time'$
  - theory elements are fixed-points: $\{P \mid HT(P) = P\}$
  - give rise to complete lattices etc.
- **signature**: operators for building programs
  - $Wait(n : \mathbb{N}) \triangleq time' = time + n \land st' = st$
  - + relational operators $; , \triangleleft , \triangleright , x := v$ etc.
  - $HT$-healthy relations are closed under these operators
  - e.g. $HT(Wait(n)) = Wait(n)$
Verification

- unified language for both specification and programs
- apply automated theorem provers to verification
- **Isabelle/UTP**: putting UTP to work
- semantic embedding of UTP in Isabelle/HOL
- syntax to enable description of programs
- specify theories and mechanically prove algebraic laws
- tactics directly leveraging Isabelle’s **proof automation**
- proven laws support verification calculi
- e.g. support for reasoning about **CSP** and **Circus** processes
- all laws presented herein have been mechanically verified
theory utp_simple_time imports "../theories/utp_theory" begin

alphabet 's st_time =
  time :: nat  st :: 's

type_synonym 's time_rel = "'s st_time hrel"

definition HT :: "'s time_rel ⇒ 's time_rel" where
  [upred_defs]: "HT(P) = (P ∧ $time ≤_u $time')"

definition Wait :: "nat ⇒ 's time_rel" where
  [upred_defs]: "Wait(n) = ($time' =_u $time + «n» ∧ $st' =_u $st)"

theorem HT_idem: "HT(HT(P)) = HT(P)" by rel_auto

theorem HT_mono: "P ⊆ Q ⇒ HT(P) ⊆ HT(Q)" by rel_auto

lemma HT_Wait: "HT(Wait(n)) = Wait(n)" by (rel_auto)

lemma HT_seqr_closed:
  "[ P is HT; Q is HT ] ⇒ P ;; Q is HT" by (rel_auto, meson dual_order.trans) -- {* Sledgehammer required *}

theorem Wait_skip: "Wait(0) = II" by (rel_auto)

theorem Wait_Wait: "Wait(m) ;; Wait(n) = Wait (m + n)" by (rel_auto)
subsection {" Actions "}

text {*} The Pay action describes the protocol when a payment of $n$ is requested between two cards, $i$ and $j$. It is slightly modified from the paper, as we firstly do not use operations but effect the transfer using indexed assignments directly, and secondly because before the transfer can proceed we need to check the balance is both sufficient, and that the transfer amount is greater than 0. It should also be noted that the indexed assignments give rise to preconditions that the list is defined at the given index. In other words, the given card records must be present. *}

definition Pay :: "index ⇒ index ⇒ money ⇒ action_mdx" where
"Pay i j n =
  pay.(«i»).(«j»).(«n») →
  ((reject.(«i») → Skip)
   ∧ «i» =u «j» ∨ «n» ≤u 0 ∨ «n» >u &valueseq(«i»)u ≥R
   &valueseq[«i»] :⇒c ( &valueseq(«i»)u - «n» ) ;
   &valueseq[«j»] :⇒c ( &valueseq(«j»)u + «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 = (∏ (i, j, n) ∈ PaySet cardNum \& Pay i j n)"

text {*} The Cycle action just repeats the payments over and over for any extant and different card indices. In order to be well-formed we require that $cardNum \ge 2$. *}
Outline

Background

Reactive Systems

Unifying Reactive and Hybrid

Towards Simulink Block Semantics

Conclusion
Reactive Processes

- UTP theory for description of reactive programs
- Basis of languages such as CSP and Circus
- Imperative programs have initial and final states
- Reactive programs additionally have intermediate states
- Intermediate programs await interaction with the environment
- Observational variables:
  - $\text{wait}, \text{wait}' : \mathbb{B}$ – distinguish intermediate and final states
  - $\text{tr}, \text{tr}' : \text{seq Event}$ – discrete history of interaction
- Portion of trace contributed by present process: $\text{tt} \triangleq \text{tr}' - \text{tr}$

Example: $\text{do}(a) \triangleq \text{II} \triangleleft \text{wait} \triangleright (\text{tr}' = \text{tr} \wedge \langle a \rangle \wedge \neg \text{wait}')$
Reactive Healthiness Conditions

\( R1(P) \equiv P \land tr \leq tr' \)

\( R2_c(P) \equiv P[\epsilon, tt/ tr, tr'] \odot tr \leq tr' \odot P \)

\( R3(P) \equiv \exists \odot wait \odot P \)

\( R \equiv R3 \circ R2_c \circ R1 \)

- **R1**: trace is monotonically increasing
- **R2_c**: no dependence on trace history
- **R3**: if predecessor is waiting then do nothing
- **R** is closed under relational calculus operators (e.g. ; and \( \square \))
Reactive Design Contracts

- building on reactive processes we have reactive designs
- \( ok, \, ok' : \mathbb{B} \) distinguish possible erroneous behaviour
- a reactive design is a triple: \([pre \mid peri \mid post]\)
  - precondition \((pre)\) is a predicate encoding assumptions on state and environment required for correct execution
  - pericondition \((peri)\) encodes commitments on trace that are satisfied by all intermediate states
  - postcondition \((post)\) encodes commitments on trace and variables that are satisfied in all final states
- contractual specifications have a natural notion of refinement:
  \[
  P_1 \Rightarrow Q_1 \quad Q_2 \land P_1 \Rightarrow P_2 \quad Q_3 \land P_1 \Rightarrow P_3 \\
  [P_1 \mid P_2 \mid P_3] \sqsubseteq [Q_1 \mid Q_2 \mid Q_3]
  \]
- objective: use reactive designs to encode Simulink blocks
Reactive Design Examples

- CSP examples use $\textit{ref}' : \mathbb{P}\textit{Event}$ to encode refusals

\[
\begin{align*}
    a \rightarrow \textit{Skip} &= \begin{cases} 
        \text{true} & | \ a \not\in \textit{ref}' \land \textit{tt} = \langle \rangle \mid \textit{st}' = \textit{st} \land \textit{tt} = \langle a \rangle \\
    \text{Stop} &= \begin{cases} 
        \text{true} & | \ \textit{tt} = \langle \rangle \mid \text{false} \\
    \end{cases} \\
    a \rightarrow \textit{Chaos} \quad \square \quad b \rightarrow \textit{Skip} &= \\
    \begin{cases} 
        \neg(\langle a \rangle \leq \textit{tt}) & | \ \textit{tt} = \langle \rangle \land a \not\in \textit{ref}' \land b \not\in \textit{ref}' \mid \textit{tt} = \langle b \rangle \land \textit{st}' = \textit{st} \\
    \end{cases} \end{align*}
\]

\[
\begin{align*}
    \textit{Chaos} &= \begin{cases} 
        \text{false} & | \ \text{false} \mid \text{false} \\
    \end{cases} \\
    \mathcal{II}_R &= \begin{cases} 
        \text{true} & | \ \text{false} \mid \mathcal{II} \\
    \end{cases} 
\end{align*}
\]
In this section, we explore the laws of reactive designs, which are fundamental to understanding how systems interact and evolve over time. These laws are crucial for developing robust and predictable cyber-physical systems. Let's delve into the core concepts:

1. **Intersection of Designs**:
   
   \[ [P_1 | P_2 | P_3] \cap [Q_1 | Q_2 | Q_3] = [P_1 \land Q_1 | P_2 \lor Q_2 | P_3 \lor Q_3] \]

2. **Chaos Law**:
   
   \[ \text{Chaos} \cap P = \text{Chaos} \]

3. **Product of Designs**:
   
   \[ \prod_{i \in I} [P_1(i) | P_2(i) | P_3(i)] = \left[ \bigwedge_{i \in I} P_1(i) \bigg| \bigvee_{i \in I} P_2(i) \bigg| \bigvee_{i \in I} P_3(i) \right] \]

4. **Initial Design**:
   
   \[ [P_1 | P_2 | P_3] ; [Q_1 | Q_2 | Q_3] = [P_1 \land P_3 \wp Q_1 | P_2 \lor P_3 ; Q_2 | P_3 ; Q_3] \]

5. **Identity Law**:
   
   \[ \mathbb{I}_R ; P = P ; \mathbb{I}_R = P \]

6. **False Design**:
   
   \[ [P_1 | P_2 | \text{false}] ; Q = [P_1 | P_2 | \text{false}] \]

7. **Chaos Law**:
   
   \[ \text{Chaos} ; P = \text{Chaos} \]

8. **Generalized Design**:
   
   \[ [P \mid Q \mid R]^{n+1} = \left[ \bigwedge_{i \leq n} (R^i \wp P) \bigg| \left( \bigvee_{i \leq n} R^i \right) ; Q \bigg| R^{n+1} \right] \]
Parallel composition

- defined wrt. a merge predicate $M$
- describes how state and traces should be merged
- rather complicated...

\[
[ P_1 \parallel P_2 \parallel P_3 ] \parallel^R_M [ Q_1 \parallel Q_2 \parallel Q_3 ] =
\]

\[
[ P_2 \ wr^*_M Q_1 \land P_3 \ wr^*_M Q_1 \land Q_2 \ wr^*_M P_1 \land Q_3 \ wr^*_M P_1 \\
| P_2 \ parallel^E_M Q_2 \land P_3 \ parallel^E_M Q_2 \land P_2 \ parallel^E_M Q_3 \\
| P_3 \ parallel^E_M Q_3 ]
\]

- $P \ wr^*_M Q$ is the weakest assumption under which process $Q$ will not violate condition $P$
- $\parallel^E_M$ merges only traces; $\parallel^*_M$ merges both states and traces
- parallel composition is always monotonic
Verification of reactive designs

- involves solving a conjecture: \([ P_1 \mid P_2 \mid P_3 ] \subseteq \text{System}\)
- \([ P_1 \mid P_2 \mid P_3 ]\): contract with assumptions and commitments

1. perform algebraic simplification of model
2. calculate pre-, peri-, and postconditions of System
3. apply refinement law to yield three proof obligations
4. discharge (or refute) POs using Isabelle tactics

- we’ve implemented tactics to facilitate this
- rdes-calc – calculate a reactive design
- rel-auto – solve relational calculus
- + Isabelle provides access to ATPs with sledgehammer
Outline

Background

Reactive Systems

Unifying Reactive and Hybrid

Towards Simulink Block Semantics

Conclusion
Hybrid Computation

- How to use reactive contracts to specify **hybrid systems**?
- Need to augment relational calculus with **continuous variables**
- I.e. piecewise continuous functions $x : \mathbb{R}_{\geq 0} \rightarrow \mathbb{R}$
- Desire to support specification in style of **duration calculus**
- Additional operators for constructing ODEs and DAEs
- We achieve this by generalising the trace model $(tr, tr')$
- Will enable a contractual approach for **Simulink**
- Inspirations:
  - Hybrid CSP (He, Zhan et al.) — DAEs and pre-emption
  - HRML (He) — tri-partite alphabet
  - Duration Calculus (Zhou et al.) — interval operator
  - Timed Reactive Designs (Hayes et al.) — timed trace model
Trace algebra

- we first abstractly characterise the trace operators

<table>
<thead>
<tr>
<th>Definition (Trace algebra)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A trace algebra $(\mathcal{T}, \wedge, \epsilon)$ is a <strong>cancellative monoid</strong> satisfying:</td>
</tr>
</tbody>
</table>

\[
x \wedge (y \wedge z) = (x \wedge y) \wedge z \quad \text{(TA1)}
\]
\[
\epsilon \wedge x = x \wedge \epsilon = x \quad \text{(TA2)}
\]
\[
x \wedge y = x \wedge z \Rightarrow y = z \quad \text{(TA3)}
\]
\[
x \wedge z = y \wedge z \Rightarrow x = y \quad \text{(TA4)}
\]
\[
x \wedge y = \epsilon \Rightarrow x = \epsilon \quad \text{(TA5)}
\]

- cancellation laws (TA3, TA4) allow us to decompose a trace
Trace operators

Definition (Trace prefix and subtraction)

\[ x \leq y \iff \exists z \bullet y = x \uplus z \]

\[ y - x \triangleq \begin{cases} 
\iota z \bullet y = x \uplus z & \text{if } x \leq y \\
\epsilon & \text{otherwise}
\end{cases} \]

- **trace prefix** \((x \leq y)\): there is a \(z\) that extends \(x\) to yield \(y\)
- **trace minus** \((y - x)\): if \(y \leq x\) then obtain the difference
## Trace prefix laws

**Theorem (Trace prefix laws)**

\[(T, \leq) \text{ is a partial order}\] 

- \(\epsilon \leq x\) \hspace{1cm} \text{(TP1)}
- \(x \leq x \uprime y\) \hspace{1cm} \text{(TP2)}
- \(x \uprime y \leq x \uprime z \iff y \leq z\) \hspace{1cm} \text{(TP4)}

**Theorem (Trace subtraction laws)**

\[x - \epsilon = x\] \hspace{1cm} \text{(TS1)}

\[\epsilon - x = \epsilon\] \hspace{1cm} \text{(TS2)}

\[x - x = \epsilon\] \hspace{1cm} \text{(TS3)}

\[(x \uprime y) - x = y\] \hspace{1cm} \text{(TS4)}

\[(x - y) - z = x - (y \uprime z)\] \hspace{1cm} \text{(TS5)}
Generalised Reactive Designs

- redefine healthiness condition $R$ using trace algebra
- define operators in terms of abstract trace behaviours
- reprove the laws of reactive designs in general context
- gives rise to generalised reactive design contracts
- different models support different reactive languages
- e.g. sequences form a trace algebra $\Rightarrow$ Circus
- allows import of law library into specialised theory
Continuous Time Traces

- model for hybrid systems is piecewise continuous functions
- finite number of left-closed/right-open continuous segments
- adapted from the work of (Hayes, 2006)
**Mathematical Model**

**Definition (Timed traces)**

\[ \mathcal{T} \mathcal{T} \triangleq \left\{ f : \mathbb{R}_{\geq 0} \to \Sigma \mid \exists t \cdot \text{dom}(f) = [0, t) \land t > 0 \Rightarrow \exists I : \mathbb{R}_{\text{oseq}} \right. \]

\[ \left. \quad \left( \text{ran}(I) \subseteq [0, t] \land \{0, t\} \subseteq \text{ran}(I) \land \left( \forall n < \#I - 1 \cdot f \text{ cont-on } [I_n, I_{n+1}) \right) \right) \right\} \]

\[ \mathbb{R}_{\text{oseq}} \triangleq \{ x : \text{seq} \mathbb{R} \mid \forall n < \#x - 1 \cdot x_n < x_{n+1} \} \]

\[ f \text{ cont-on } [m, n) \triangleq \forall t \in [m, n) \cdot \lim_{x \to t} f(x) = f(t) \]

- \( I \): a finite sequence of continuous intervals
- \( \Sigma \): a **topological space** denoting the continuous state (e.g. \( \mathbb{R}^n \))
- Continuous variables are projections (**lenses**) on \( \Sigma \)
### Operators

**Definition (Timed-trace operators)**

\[
\begin{align*}
    f \gg n & \triangleq \lambda x \cdot f(x - n) \\
    \text{end}(f) & \triangleq \min(\mathbb{R}_{\geq 0} \setminus \text{dom}(f)) \\
    \langle \rangle & \triangleq \emptyset \\
    f \bowtie g & \triangleq f \cup (g \gg \text{end}(f))
\end{align*}
\]

- \( f \gg n \) shifts \( f : \mathbb{R} \to A \) to the right by \( n : \mathbb{R} \)
- \( \text{end}(f) \) obtains the limit point of a trace
- **Theorem**: \((\mathcal{TT}, \bowtie, \langle \rangle)\) forms a trace algebra
- **Key Lemma**: \(\mathcal{TT}\) is closed under \(\bowtie\)
- We have now obtained a model of **hybrid reactive processes**
Hybrid relational calculus

- kernel language of relational hybrid programs
- augments relational calculus with continuous variables ($x$)
- linked to discrete copies via coupling invariants
- continuous alphabet: $\text{con}_\alpha(P)$, discrete alphabet: $\text{dis}_\alpha(P)$
- operators denoted in terms of hybrid reactive processes
Hybrid relational calculus

- kernel language of relational hybrid programs
- augments relational calculus with continuous variables ($x$)
- linked to discrete copies via coupling invariants
- continuous alphabet: $\text{con}_\alpha(P)$, discrete alphabet: $\text{dis}_\alpha(P)$
- operators denoted in terms of hybrid reactive processes
- retain standard relational operators
  - composition ($;$), assignment ($x := v$), if-then-else etc.
Hybrid relational calculus

- kernel language of relational hybrid programs
- augments relational calculus with continuous variables ($x$)
- linked to discrete copies via coupling invariants
- continuous alphabet: $\text{con}\alpha(P)$, discrete alphabet: $\text{dis}\alpha(P)$
- operators denoted in terms of hybrid reactive processes
- retain standard relational operators
  - composition ($;$), assignment ($x := v$), if-then-else etc.
- add continuous evolution operators
  - differential algebraic equations — $\langle \dot{v}_1 = f_1; \cdots; \dot{v}_n = f_n | B \rangle$
  - pre-emption — $P [ B ] Q$
  - interval (continuous specification) — $[P]$
Hybrid relational calculus

- kernel language of relational hybrid programs
- augments relational calculus with continuous variables ($x$)
- linked to discrete copies via coupling invariants
- continuous alphabet: $\text{con}_\alpha(P)$, discrete alphabet: $\text{dis}_\alpha(P)$
- operators denoted in terms of hybrid reactive processes
- retain standard relational operators
  - composition ($;$), assignment ($x := v$), if-then-else etc.
- add continuous evolution operators
  - differential algebraic equations — $\langle \dot{v}_1 = f_1; \cdots; \dot{v}_n = f_n \mid B \rangle$
  - pre-emption — $P[ B ] Q$
  - interval (continuous specification) — $[ P ]$
Example 1: Simple Bouncing Ball

Bouncing ball in Modelica

```model BouncingBall
  Real p(start=2, fixed=true), v(start=0, fixed=true);
  equation
  der(v) = -9.81;
  der(p) = v;
  when p <= 0 then
    reinit(v, -0.8*v);
  end when;
end BouncingBall;
```

Bouncing ball in hybrid relational calculus

\[ p, v := 2, 0 ; (\langle \dot{p} = v; \dot{v} = -9.81 \rangle [ p \leq 0 ] v := -v * .8)\]
Hybrid denotational semantics

\[ \tt \triangleq \tr' - \tr \]
\[ \underline{x}(t) \triangleq \tt(t) \cdot x \]
\[ \ell \triangleq \text{end}(\tt) \]
\[ P \@ \tau \triangleq \{ \underline{x} \mapsto \tt(\tau) \cdot x \mid x \in \text{con}_\alpha(P) \setminus \{t\} \} \upharpoonright P \]

- \( \sigma \upharpoonright P \) applies substitution function \( \sigma \) to \( P \)
- \( P \@ \tau \) lifts continuous variables to instant \( \tau \)
- e.g. \( (x > 1 \land y = x \cdot 3) \@ \tau = (x(\tau) > 1 \land y(\tau) = x(\tau) \cdot 3) \)
- borrowed from \textit{timed refinement calculus}
Interval operator

\[ [P(\tau)] \triangleq tr' \geq tr \land (\forall t \in [0, \ell) \bullet P(t) \circ t) \]

- cf. Duration Calculus (Zhou, Ravn, and Hanzen, 1993)
- continuous spec: states that \( P \) holds on the interval \([0, \ell)\)
- e.g. \([15 < temp \land temp \leq 30] \)

Theorem (Interval laws)

\[
\begin{align*}
[P(\tau) \land Q(\tau)] &= [P(\tau)] \land [Q(\tau)] \\
[P(\tau) \lor Q(\tau)] &\subseteq [P(\tau)] \lor [Q(\tau)] \\
[true] &= R1(true) \\
[false] &= (tr' = tr) \\
(\forall \tau \bullet P(\tau) \Rightarrow Q(\tau)) &\Rightarrow [Q(\tau)] \subseteq [P(\tau)]
\end{align*}
\]
Hybrid specification

$$\llbracket P \rrbracket \triangleq \llbracket P \rrbracket \land \ell > 0 \land \bigwedge_{v \in \text{con}_\alpha(P)} \left( v = v(0) \land v' = \lim_{t \to \ell} v(t) \right) \land \Pi_{\text{dis}_\alpha}(P)$$

- continuous state evolves according to $P$
- must make non-zero progress to avoid certain Zeno effects
- initial condition for $v$ taken from discrete copy
- final condition for $v$ taken from limit construction
Hybrid specification

\[ \llbracket P \rrbracket \triangleq [P] \land \ell > 0 \land \bigwedge_{v \in \text{con}_\alpha(P)} \left( v = v(0) \land v' = \lim_{t \to \ell} (v(t)) \right) \land \mathbb{II}_{\text{dis}_\alpha(P)} \]

- continuous state evolves according to \( P \)
- must make non-zero progress to avoid certain Zeno effects
- initial condition for \( v \) taken from discrete copy
- final condition for \( v \) taken from limit construction
Hybrid Evolution

\[
\langle \dot{x} = F'(x, \dot{x}) \mid B(x) \rangle \triangleq \exists F \bullet \left[ F' \text{ has-deriv } F \text{ at } \tau \land x = F(\tau) \land B(x) \right]
\]

\[
P[ B ] Q \triangleq (Q < B @ 0 \triangleright (P \land \neg B)) \lor ((P \land \neg B \land B') ; Q)
\]

- **differential-algebraic equation:** \( \langle \dot{x} = F'(x, \dot{x}) \mid B(x) \rangle \)
- there exists a solution \( (F) \) satisfying the ODE and constraint
- **pre-emption:** \( P[ B ] Q \)
- \( P \) evolves until \( B \) becomes true, then \( Q \) is enabled
- **remember:** all constructs boil down to constraints on the continuous trace
INtegrated TOolchain for Cyber-Physical Systems

Mechanisation in Isabelle/UTP

- based on Multivariate Analysis and HOL-ODE packages
- support for limits, ODEs, DAEs, and their solutions
- solutions can be verified but not generated

```plaintext
abbreviation "grav" == -9.81

definition "bouncing_ball =  
(\mu x \cdot c:vel, c:pos := 0.2.0 ;; 
(pos,vel \cdot \langle (\lambda t (v,p). (grav, v)) \rangle_H 
[&pos <= u 0]_H 
(c:vel := -0.8 * &c:vel ;; X))")

definition "thermostat =  
(\mu x \cdot c:temp, d:isOn := 20, false ;; 
((temp \cdot \langle (\lambda _ t. 5 - 0.1 * t) \rangle_H \& d:isOn \& (temp \cdot \langle (\lambda _ t. - 0.1 * t) \rangle_H) 
[&temp < u 19 \& temp > u 21]_H 
(d:isOn := true \& c:temp < u 19 \& d:isOn := false))"

consts 
thermostat :: 
((\'_a thermostat_c_scheme ttrace,  
(\'_b thermostat_d_scheme \& \'_a thermostat_c_scheme) rsp_vars) rp,
```

http://into-cps.au.dk/
Outline

Background

Reactive Systems

Unifying Reactive and Hybrid

Towards Simulink Block Semantics

Conclusion
Recap

- we have presented reactive design contracts
- with an intuitive notion of refinement
- we then generalised the underlying trace semantics
- new model based on piecewise-continuous functions
- and constructed a hybrid relational calculus
- we can now combine these to describe Simulink blocks
Overview

- each block is a parametric reactive design
- parameters are constants and shared wires
- wires are modelled as continuous variables
- blocks constrain the behaviours of the wires
- preconditions prevent erroneous behaviour (e.g. div by zero)
- parallel compose instantiated blocks to give overall diagram
- yields a system of differential and algebraic equations
Simulink blocks (tentative!)

- **Source**($x, v$) $\cong [true \mid x = v \mid false]$
- **Term**($x$) $\cong [true \mid true \mid false]$
- **Add**($x, y, z$) $\cong [true \mid z = x + y \mid false]$
- **Gain**($x, y, v$) $\cong [true \mid y = v \cdot x \mid false]$
- **Divide**($x, y, z$) $\cong [y \neq 0 \mid z = x/y \mid false]$
- **Integrate**($x, y, y_0, r$) $\cong x := x_0 ; (\langle \dot{y} = x \rangle [r] x := x_0)^\omega$
- **InitVal**($x, y, v$) $\cong [true \mid y = v \triangleright \tau = 0 \triangleright y = x \mid false]$
- **Cond**($x, y, f$) $\cong [true \mid y = f(x) \mid false]$

- all instances of general pattern: **Init** ; (**Cont** [**Cond**] **Disc)**$^\omega$
Bouncing Ball Example

from https://uk.mathworks.com/help/simulink/examples/simulation-of-a-bouncing-ball.html
Translation

- we need to give each wire an identifier in the alphabet
- alphabet: $\underline{g}, \underline{v}, \underline{v_0}, \underline{v_1}, \underline{p}, \underline{p_0}, \underline{p_1}: \mathbb{R}$ and $\underline{r}: \mathbb{B}$
- diagram described by following parallel composition of blocks:

\[
\text{Source}(\underline{g}, -9.81) \parallel \text{Integrate}(\underline{g}, \underline{v_0}, \underline{v}, \underline{r}) \parallel \text{Gain}(\underline{v}, \underline{v_1}, -0.8) \\
\parallel \text{InitVal}(\underline{v_1}, \underline{v_0}, 15) \parallel \text{Integrate}(\underline{v}, \underline{p}, \underline{p_0}, \underline{r}) \parallel \text{Source}(\underline{p_1}, 0) \\
\parallel \text{InitVal}(\underline{p_1}, \underline{p_0}, 10) \parallel \text{Cond}(\underline{p}, \underline{r}, \lambda x \bullet x \leq 0) \parallel \text{Term}(\underline{p})
\]

- verification requires calculation of pre- and periconditions
- this involves flattening and solving the system of equations
Outline

Background

Reactive Systems

Unifying Reactive and Hybrid

Towards Simulink Block Semantics

Conclusion
Conclusion

- reactive designs facilitate contractual specifications
- Isabelle/UTP provides verification infrastructure
- generalised trace model facilitates hybrid models
- combining these provides denotational foundations for Simulink
- UTP enables combination of models from different languages
  - programming languages (for controller implementation)
  - modelling languages (Modelica, SysML etc.)
- enable a multi-disciplinary approach to system design
- could apply to justify optimisations and transformations
Future work

- investigate correct merge predicate for Simulink
- lots more algebraic laws to prove
- alternative trace models (superdense-time?)
- automated flattening of Simulink diagrams
- reasoning about ODEs in Isabelle/HOL (incl. approximations)
  - cf. work of Fabian Immler and Johannes Hölzl
- integration of CAS with Isabelle
- static analysis of Simulink diagrams
- combination with existing work with Circus
- unifying with our Modelica semantics
References

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