What I did last Summer

Andrew Butterfield

Trinity College, University of Dublin

Friday Get-Together

November 30th 2007

TCD

Introduction

- Context
  - Developing Theories as part of UTP (Unifying Theories of Programming)
  - Predicates relating pre- and post-observations
  - Notion of Healthy Predicates (realistic, feasible, desirable, practical)
  - Interestin so-called “Reactive Systems” (concurrent/event-driven)
- Issue
  - Long tedious proofs
  - Logic needs to be 2nd-order (at least)
  - Specific handling of undefinedness

Reactive Systems

To model reactive systems (a.k.a. "processes") we need to track four observations:

ok	:	$\mathbb{B}$	— process is stable (not diverging)
$wait$	:	$\mathbb{B}$	— process is waiting for an event, and has not terminated
tr	:	$Event^*$	— event history
ref	:	$\mathbb{P}\ Event$	— events being refused

We define predicates that *relate* the before-state ($ok, wait, tr, ref$) to the after-state ($ok', wait', tr', ref'$) of a process (Relational Semantic Model).

The language used to describe processes is very CSP-like.

Examples

- A process that performs event *a* and then terminates ($a \longrightarrow SKIP$)

$$ok' \land \neg\, wait' \land tr' = tr \frown \langle a \rangle$$

- A process that performs *a* and then behaves like process *P* ($a \longrightarrow P$)

$$(ok' \land \neg\, wait' \land tr' = tr \frown \langle a \rangle) \fatsemi P$$

(We use *P* to signify the process, and its predicate (“programs are predicates” - Hehner))

- The definition of sequential composition:

$$\begin{array}{lcl} P \fatsemi Q & \widehat{=} & \exists\, ok_0, wait_0, tr_0, ref_0 \bullet \\ & & \quad P[ok_0, wait_0, tr_0, ref_0 / ok', wait', tr', ref'] \\ & & \quad \land \\ & & \quad Q[ok_0, wait_0, tr_0, ref_0 / ok, wait, tr, ref] \end{array}$$

(Relational Composition)

Bad Examples

Unfortunately we can also write predicates that are not sensible:

- Messing with time (unrealistic, infeasible):

$$tr = tr' \frown \langle a_1, \ldots, a_n \rangle$$

$$wait \wedge \neg\, wait'$$

- Arbitrary knowledge of/restrictions on past history (infeasible, impractical):

$$\textbf{if } tr = \langle a_1, \ldots, a_n \rangle \textbf{ then } P \textbf{ else } Q$$

$$tr = \langle a, b, c \rangle \wedge tr' = \langle a, b, c, d, e \rangle$$

- Specifying Bad Things (undesirable)

$$\neg\, ok'$$

We use a mechanism called *Healthiness Conditions* to filter these out.

Introducing Healthiness

- We want to prevent nonsense like: $tr = tr' \frown \ldots$
- It seems reasonable that a healthy predicate entails $tr \leq tr'$ (prefix)

$$\text{Healthy } P \Rightarrow tr \leq tr'$$

- Plan: use a predicate-function (transformer!) **mkH** to make a predicate “H-healthy”.
  - Predicate function is idempotent: $\mathbf{mkH} \circ \mathbf{mkH} = \mathbf{mkH}$
  - Healthy predicates are fixed-points of the predicate-function: $\mathbf{isH}(P) \mathrel{\widehat{=}} P \equiv \mathbf{mkH}(P)$
- In UTP, it is usual to refer to both **mkH** and **isH** as simply **H**.

Introducing R1

- We say a predicate is Reactive-1 (**R1**) Healthy if the trace is only extended:
- Looking at what is required:

$$
\begin{array}{ll}
 & \mathbf{isR1}(P) \\
\equiv & \text{“ key property we want ”} \\
 & P \Rightarrow tr \le tr' \\
\equiv & \text{“ propositional calculus ”} \\
 & P \equiv P \wedge tr \le tr'
\end{array}
$$

- Introducing **R1**:

$$
\begin{array}{rcl}
GROW & \widehat{=} & tr \le tr' \\
\mathbf{mkR1}(P) & \widehat{=} & P \wedge GROW \\
\mathbf{isR1}(P) & \widehat{=} & P \equiv \mathbf{mkR1}(P)
\end{array}
$$

R1 is idempotent

$$
\begin{array}{ll}
 & \mathbf{R1}\big(\mathbf{R1}(P)\big) \\
\equiv & \quad \text{“ defn. } \mathbf{R1}\text{, twice ”} \\
 & \big(P \wedge \mathit{GROW}\big) \wedge \mathit{GROW} \\
\equiv & \quad \text{“ } \wedge\text{-assoc, -idem. ”} \\
 & P \wedge \mathit{GROW} \\
\equiv & \quad \text{“ defn } \mathbf{R1}\text{, backwards ”} \\
 & \mathbf{R1}(P)
\end{array}
$$

More Healthiness

- A Process is **R2**-healthy if it's behaviour does not depend on *tr* (past event history)

$$\mathbf{R2}(P) \mathrel{\widehat{=}} \exists\, s \bullet P[s, s \frown (tr' - tr)/tr, tr']$$

- A Process is **R3**-healthy if it specifies that nothing changes if it hasn't started (provided the previous process is not diverging).

$$\begin{array}{rcll} DIV & \widehat{=} & \neg\, ok \wedge GROW & \text{— divergence} \\ STET & \widehat{=} & wait' = wait \wedge tr' = tr \wedge ref' = ref & \text{— no change} \\ \mathit{I\!I} & \widehat{=} & DIV \vee ok' \wedge STET & \\ \mathbf{R3}(P) & \widehat{=} & \mathit{I\!I} \lhd wait \rhd P & \end{array}$$

- A process is Reactive-Healthy if it is **R1**-, **R2**- and **R3**-healthy

$$\mathbf{R} \quad \widehat{=} \quad \mathbf{R3} \circ \mathbf{R2} \circ \mathbf{R1}$$

Commuting Healthiness

- Why did we compose in the order we did ?

$$
\begin{aligned}
\mathbf{R} &\mathrel{\widehat{=}} \mathbf{R3} \circ \mathbf{R2} \circ \mathbf{R1} \\
&=? \ \mathbf{R2} \circ \mathbf{R3} \circ \mathbf{R1} \\
&=? \ \mathbf{R2} \circ \mathbf{R1} \circ \mathbf{R3} \\
&=? \ \mathbf{R1} \circ \mathbf{R2} \circ \mathbf{R3} \\
&=? \ \mathbf{R1} \circ \mathbf{R3} \circ \mathbf{R2} \\
&=? \ \mathbf{R3} \circ \mathbf{R1} \circ \mathbf{R2}
\end{aligned}
$$

- It is (*very*) useful to have healthiness conditions that commute:

$$\mathbf{R1} \circ \mathbf{R2} = \mathbf{R2} \circ \mathbf{R1} \qquad \mathbf{R1} \circ \mathbf{R3} = \mathbf{R3} \circ \mathbf{R1} \qquad \mathbf{R3} \circ \mathbf{R2} = \mathbf{R2} \circ \mathbf{R3}$$

Ideally these will be theorems.

Undefinedness

Undefinedness plays a role in these healthiness, conditions, particularly with **R2**.

$$\exists s \bullet P[s, s \frown (tr' - tr)/tr, tr']$$

What happens if $tr \not\leq tr'$?

We attempt to prove that

$$\mathbf{R1} \circ \mathbf{R2} = \mathbf{R2} \circ \mathbf{R1}$$

Proof that R1 and R2 commute

$$
\begin{array}{ll}
 & \mathbf{R2}(\mathbf{R1}(P)) \\
\equiv & \quad \text{“ defn. } \mathbf{R1} \text{ ”} \\
 & \mathbf{R2}(P \wedge tr \leq tr') \\
\equiv & \quad \text{“ defn. } \mathbf{R2} \text{ ”} \\
 & \exists\, s \bullet (P \wedge tr \leq tr')[s, s \frown (tr' - tr)/tr, tr'] \\
\equiv & \quad \text{“ apply substitution ”} \\
 & \exists\, s \bullet P[s, s \frown (tr' - tr)/tr, tr'] \wedge s \leq s \frown (tr' - tr) \\
\equiv & \quad \text{“ ??? is } s \leq s \frown (tr' - tr) \equiv \textbf{true} \text{ ? ”} \\
 & ????
\end{array}
$$

Proof that R1 and R2 don't commute

$$\mathbf{R2}(\mathbf{R1}(P))$$

$\equiv$ " defn. **R1** "

$$\mathbf{R2}(P \land tr \leq tr')$$

$\equiv$ " defn. **R2** "

$$\exists\, s \bullet (P \land tr \leq tr')[s, s \frown (tr' - tr)/tr, tr']$$

$\equiv$ " apply substitution "

$$\exists\, s \bullet P[s, s \frown (tr' - tr)/tr, tr'] \land s \leq s \frown (tr' - tr)$$

$\equiv$ " $s \leq s \frown \_ \equiv \mathbf{true}$ "

$$\exists\, s \bullet P[s, s \frown (tr' - tr)/tr, tr']$$

$\equiv$ " defn. **R2** "

$\mathbf{R2}(P)$!!!!

Proof that R1 and R2 do commute

$$\mathbf{R2}(\mathbf{R1}(P))$$

$\equiv$ " defn. **R1** "

$$\mathbf{R2}(P \land tr \le tr')$$

$\equiv$ " defn. **R2** "

$$\exists s \bullet (P \land tr \le tr')[s, s \frown (tr' - tr)/tr, tr']$$

$\equiv$ " apply substitution "

$$\exists s \bullet P[s, s \frown (tr' - tr)/tr, tr'] \land s \le s \frown (tr' - tr)$$

$\equiv$ " $s \le s \frown (tr' - tr) \equiv tr \le tr'$ "

$$\exists (s \bullet P[s, s \frown (tr' - tr)/tr, tr'] \land tr \le tr')$$

$\equiv$ " shrink scope "

$$\exists (s \bullet P[s, s \frown (tr' - tr)/tr, tr']) \land tr \le tr'$$

$\equiv$ " defn. **R2**, **R1** "

$$\mathbf{R1}(\mathbf{R2}(P))$$

The Choice of Logic Does Matter !

- If we want **R1** and **R2** to commute, we must use a specific logic variant
- Semi-Classical Logic
  - Predicates are two-valued
  - Expression can be undefined, but this does not leak up to the Predicate level.
  - As used in Z
- We have predicate-functions, and recursion requires us to quantify over predicates, so logic needs to be 2nd-order.

The Truth regarding $s \leq s \frown (tr' - tr)$

- In semi-classical logic, we require all terms/sub-terms to be defined ($\mathcal{D}$):

$$s \leq (s \frown t) \equiv \mathcal{D}(s) \wedge \mathcal{D}(t)$$

Variables are always defined (we only quantify over defined values), to we can deduce:

$$\begin{aligned} s \leq s \frown (tr' - tr) &\equiv \mathcal{D}(s) \wedge \mathcal{D}(tr' - tr) \\ &\equiv tr \leq tr' \end{aligned}$$

- Other logics (three-valued, or based on a notion of computation) may capture the notion that we don't need to know the value of *t* in order to show the truth of the above.

$$s \leq s \frown \_ \equiv \textbf{true}$$

Why not use an existing higher-order prover?

- PVS
  - total functions, so need to model undefinedness explicitly
- Isabelle/HOL
  - unclear how to embed own logic (possible, I know, but unclear what is involved)
  - has explicit embedding of own logic into ML-like metalanguage
- CoQ
  - Curry-Howard Isomorphism is cool, but . . .
  - also has a Totality Requirement
  - Need to jump through “direct-sum” hoops to do simple proofs

Also, I prefer to see steps of a proof, rather than a list of tactics, as the final transcript

Introducing Saothín

- Proof Assistant (2nd-Order, semi-classical)
- Implemented in Haskell
  - uses wxHaskell for GUI
  - runs on Windows (98, XP, Vista)
  - should run on Linux, Mac OS X
- See `www.cs.tcd.ie/Andrew.Butterfield/Saothin/`

Doing Formal Methods

Thesis:

“ To *do* formal methods, one should *implement* a theorem prover ”

Antithesis:

“you have not proved anything with your theorem prover until you have proved the theorem prover correct!”

Discuss.

Thank you for your kind attention

$$(ok', \neg\ wait', thirsty')$$