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:

\[ \begin{align*}
ok & : \mathbb{B} & \quad \text{— process is stable (not diverging)} \\
wait & : \mathbb{B} & \quad \text{— process is waiting for an event, and has not terminated} \\
tr & : Event^* & \quad \text{— event history} \\
ref & : \mathbb{P} Event & \quad \text{— events being refused}
\end{align*} \]

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 ($a \rightarrow SKIP$)
  
  \[
  ok' \land \neg wait' \land tr' = tr \land \langle a \rangle
  \]

- A process that performs $a$ and then behaves like process $P$ ($a \rightarrow P$)
  
  \[
  (ok' \land \neg wait' \land tr' = tr \land \langle a \rangle) \circ P
  \]
  
  (We use $P$ to signify the process, and its predicate (“programs are predicates” - Hehner))

- The definition of sequential composition:
  
  \[
  P \circ Q \equiv \exists ok_0, wait_0, tr_0, ref_0 \bullet
  
  \begin{align*}
  &P[ok_0, wait_0, tr_0, ref_0 / ok', wait', tr', ref'] \\
  \land \\
  &Q[ok_0, wait_0, tr_0, ref_0 / ok, wait, tr, ref]
  \end{align*}
  \]
  
  (Relational Composition)
Bad Examples

Unfortunately we can also write predicates that are not sensible:

- Messing with time (unrealistic, infeasible):

  \[ tr = tr' \land \langle a_1, \ldots, a_n \rangle \]
  \[ wait \land \neg wait' \]

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

  \[
  \text{if } tr = \langle a_1, \ldots, a_n \rangle \text{ then } P \text{ else } Q \\
  tr = \langle a, b, c \rangle \land tr' = \langle a, b, c, d, e \rangle
  \]

- 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: \( tr = tr' \wedge \ldots \)

- 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!) \( \text{mkH} \) to make a predicate “H-healthy”.
  - Predicate function is idempotent: \( \text{mkH} \circ \text{mkH} = \text{mkH} \)
  - Healthy predicates are fixed-points of the predicate-function: \( \text{isH}(P) \equiv P \equiv \text{mkH}(P) \)

- In UTP, it is usual to refer to both \( \text{mkH} \) and \( \text{isH} \) 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:

\[ \text{isR1}(P) \]
\[ \equiv \quad \text{“ key property we want ”} \]
\[ P \Rightarrow tr \leq tr' \]
\[ \equiv \quad \text{“ propositional calculus ”} \]
\[ P \equiv P \land tr \leq tr' \]

• Introducing R1:

\[ \text{GROW} \quad \triangleq \quad tr \leq tr' \]
\[ \text{mkR1}(P) \quad \triangleq \quad P \land \text{GROW} \]
\[ \text{isR1}(P) \quad \triangleq \quad P \equiv \text{mkR1}(P) \]
R1 is idempotent

\[ R1(R1(P)) \]
\[ \equiv \quad \text{“ defn. } R1, \text{ twice ”} \]
\[ (P \land GROW) \land GROW \]
\[ \equiv \quad \text{“ } \land\text{-assoc, } \land\text{-idem. ”} \]
\[ P \land GROW \]
\[ \equiv \quad \text{“ defn } R1, \text{ backwards ”} \]
\[ R1(P) \]
More Healthiness

- A Process is \( R_2 \)-healthy if it’s behaviour does not depend on \( tr \) (past event history)

\[
R_2(P) \equiv \exists s \cdot P[s, s \quad (tr' - tr)/tr, tr']
\]

- A Process is \( R_3 \)-healthy if it specifies that nothing changes if it hasn’t started (provided the previous process is not diverging).

\[
\begin{align*}
DIV & \equiv \neg ok \land GROW \quad \text{— divergence} \\
STET & \equiv \text{wait}' = \text{wait} \land tr' = tr \land ref' = ref \quad \text{— no change} \\
\Pi & \equiv DIV \lor ok' \land STET \\
R_3(P) & \equiv \Pi \triangleleft \text{wait} \triangleright P
\end{align*}
\]

- A process is Reactive-Healthy if it is \( R_1 \)-, \( R_2 \)- and \( R_3 \)-healthy

\[
R \equiv R_3 \circ R_2 \circ R_1
\]
Commuting Healthiness

• Why did we compose in the order we did?

\[ R \cong R_3 \circ R_2 \circ R_1 \]
\[ =? \quad R_2 \circ R_3 \circ R_1 \]
\[ =? \quad R_2 \circ R_1 \circ R_3 \]
\[ =? \quad R_1 \circ R_2 \circ R_3 \]
\[ =? \quad R_1 \circ R_3 \circ R_2 \]
\[ =? \quad R_3 \circ R_1 \circ R_2 \]

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

\[ R_1 \circ R_2 = R_2 \circ R_1 \quad R_1 \circ R_3 = R_3 \circ R_1 \quad R_3 \circ R_2 = R_2 \circ R_3 \]

Ideally these will be theorems.
Undefinedness plays a role in these healthiness, conditions, particularly with \( \textbf{R2} \).

\[
\exists s \bullet P[s, s \overset{s}{\sim} (tr' - tr)/tr, tr']
\]

What happens if \( tr \nleq tr' \)?

We attempt to prove that

\[
\textbf{R1} \circ \textbf{R2} = \textbf{R2} \circ \textbf{R1}
\]
Proof that R1 and R2 commute

\[ R_2(R_1(P)) \]
\[ \equiv \quad \text{"defn. } R_1 \text{"} \]
\[ R_2(P \land tr \leq tr') \]
\[ \equiv \quad \text{"defn. } R_2 \text{"} \]
\[ \exists s \cdot (P \land tr \leq tr')[s, s \triangle (tr' - tr) / tr, tr'] \]
\[ \equiv \quad \text{"apply substitution"} \]
\[ \exists s \cdot P[s, s \triangle (tr' - tr) / tr, tr'] \land s \leq s \triangle (tr' - tr) \]
\[ \equiv \quad \text{"?? is } s \leq s \triangle (tr' - tr) \equiv \text{true ?"} \]

"???"
Proof that $R_1$ and $R_2$ don’t commute

\[ R_2(R_1(P)) \]
\[ \equiv \text{ “ defn. } R_1 \text{ ”} \]
\[ R_2(P \land tr \leq tr') \]
\[ \equiv \text{ “ defn. } R_2 \text{ ”} \]
\[ \exists s \bullet (P \land tr \leq tr')[s, s \uplus (tr' - tr)/tr, tr'] \]
\[ \equiv \text{ “ apply substitution ”} \]
\[ \exists s \bullet P[s, s \uplus (tr' - tr)/tr, tr'] \land s \leq s \uplus (tr' - tr) \]
\[ \equiv \text{ “ } s \leq s \uplus - \equiv \text{ true ”} \]
\[ \exists s \bullet P[s, s \uplus (tr' - tr)/tr, tr'] \]
\[ \equiv \text{ “ defn. } R_2 \text{ ”} \]
\[ R_2(P) \quad !!!! \]
Proof that R1 and R2 do commute

\[ R2(R1(P)) \]
\[ \equiv \quad \text{"defn. } R1 \text{"} \]
\[ R2(P \land tr \leq tr') \]
\[ \equiv \quad \text{"defn. } R2 \text{"} \]
\[ \exists s \bullet (P \land tr \leq tr')[s, s \preceq (tr' - tr)/tr, tr'] \]
\[ \equiv \quad \text{"apply substitution"} \]
\[ \exists s \bullet P[s, s \preceq (tr' - tr)/tr, tr'] \land s \preceq s \preceq (tr' - tr) \]
\[ \equiv \quad \text{"} s \preceq s \preceq (tr' - tr) \equiv tr \leq tr' \text{"} \]
\[ \exists(s \bullet P[s, s \preceq (tr' - tr)/tr, tr']) \land tr \leq tr' \]
\[ \equiv \quad \text{"shrink scope"} \]
\[ \exists(s \bullet P[s, s \preceq (tr' - tr)/tr, tr']) \land tr \leq tr' \]
\[ \equiv \quad \text{"defn. } R2, R1 \text{"} \]
\[ R1(R2(P)) \]
The Choice of Logic Does Matter!

• If we want $R_1$ and $R_2$ 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 \( s \leq s \wedge (tr' - tr) \)

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

\[
s \leq (s \wedge t) \equiv \mathcal{D}(s) \wedge \mathcal{D}(t)
\]

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

\[
s \leq s \wedge (tr' - tr) \equiv \mathcal{D}(s) \wedge \mathcal{D}(tr' - tr)
\equiv tr \leq tr'
\]

- 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.

\[
s \leq s \wedge \_ \equiv \text{true}
\]
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
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

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