Behaviour Analysis
for Validating
Communication Patterns

Torben Amtoft

joint work with
Flemming Nielson   Hanne Riis Nielson

March 16, 2000

Boston University
“Programming the Internet” seminar
— Our Approach to Validating Software

Given

- A reactive system $R$
- A protocol $PC$ that $R$ must obey
- A program $P$ implementing $R$

Task:

- Ensure that $P$ obeys $PC$

Method:

- Analyze $P$ and extract succinct representation $b$
  (tuned for $PC$)
- Check that $b$ obeys $PC$
  (formally or informally)
Example

Safety protocol:

- Machine $M$ must not be started until temperature is OK

System interface:

- $M$ started by sending signal over start\_M
- a thermometer sends a signal over temp\_OK when temperature has reached the desired level

Suppose that our analyzer from $P$ extracts behaviour $b$ given by:

$$b = \cdots; \text{temp\_OK?}; \text{start\_M!}; b$$

Then protocol is satisfied
(at least wrt. high-level model)
Achievements

- Type and effect system for extracting behaviours from higher-order concurrent programs
- Inference algorithm producing manageable output
- Prototype implementation displaying communication pattern in readable form

Theoretical foundations:

- Type system sound wrt. a small-step semantics
- Algorithm (incl. simplification steps) sound wrt. type system (and also complete)
Road map

- Karlsruhe Production cell (benchmark)
- Introduce Concurrent ML, and develop program for benchmark
- Introduce behaviours
- A type and effect system extracting behaviours
- An algorithm inferring behaviours
- Methods for simplifying behaviours
- Apply prototype to example program
- Case study: apply prototype to a proposed solution to benchmark
- The prototype: interface, experiments
The Karlsruhe production cell
The Karlsruhe production cell

Benchmark (Springer LNCS 891) for designing and validating reactive systems

Safety properties include

- no blanks put on top of each other on the deposit belt
- table not moved upwards if in upper position,
  table not moved downwards if in lower position
- table alternates between clockwise and counterclockwise rotation
— Concurrent ML —

Extends SML

\[ e ::= c \mid x \mid \text{fn } x \Rightarrow e \mid e_1 \ e_2 \]
\[ \mid \text{let } x = e_1 \text{ in } e_2 \]
\[ \mid \text{rec } f \ x \Rightarrow e \mid \text{if } e \text{ then } e_1 \text{ else } e_2 \]

with concurrent constants (Reppy)

\[ c ::= \text{spawn} \mid \text{channel}^l \ (\text{dynamic topology}) \]
\[ \mid \text{send} \mid \text{accept} \ (\text{communication}) \]
\[ \mid \ldots \]

Labels on channel used to track (monovariantly) origin of channels

Key feature: events

- denote “communication possibilities”
- first-class values

7
— Example CML building block —

```haskell
fun move_until wait_fun start_ch stop_ch =
  ( send(start_ch,())
  ; wait_fun ()
  ; send(stop_ch,()) )
```

which CML assigns (higher-order) type

\[(\text{unit} \to \alpha) \to \text{unit chan} \to \text{unit chan} \to \text{unit}\]

Basic “double handshake” protocol:

```haskell
fun passive_sync ready_ch =
  ( send(ready_ch,())
  ; accept(ready_ch)
  )
```

```haskell
fun active_sync ready_ch crit_action =
  ( accept(ready_ch)
  ; crit_action ()
  ; send(ready_ch,())
  )
```
— Multiple handshake partners —

if ready ch1 \rightarrow \text{passive\_sync} \ ch1;
\quad \text{passive\_sync} \ ch2

[] ready ch2 \rightarrow \text{passive\_sync} \ ch2;
\quad \text{passive\_sync} \ ch1

This pseudo-code can be expressed in CML, using events.

Events are created by

- \text{transmit} (non-committing \text{send})
- \text{receive} (non-committing \text{accept})

Events are consumed by

- \text{sync} (converts into action)
- \text{select} (chooses feasible alternative)

\[
\begin{align*}
\text{send} \ x & \quad = \quad \text{sync} (\text{transmit} \ x) \\
\text{accept} \ x & \quad = \quad \text{sync} (\text{receive} \ x)
\end{align*}
\]
Expressing multiple handshake

wrap inlines "continuation" into event

```javascript
select [wrap( transmit(ch1,()),
    fn () => ( accept(ch1)
        ; passive_sync ch2)),
    wrap( transmit(ch2,()),
    fn () => ( accept(ch2)
        ; passive_sync ch1))]
```

Abstracting the pattern:

```javascript
select [passive_sync_event ch1
    fn () => passive_sync ch2,
    passive_sync_event ch2
    fn () => passive_sync ch1]
```

```javascript
fun passive_sync_event ready_ch cont =
    wrap ( transmit(ready_ch,()),
    fn () => ( accept(ready_ch)
        ; cont () ))
```
fun belt2 () =
    let fun belt2_cycle () =
        ( move_until (fn () => (accept(belt2_blank_at_end);
                              accept(belt2_no_blank_at_end)))
            belt2_start
            belt2_stop;
        select [passive_sync_event belt2_ready_for_arm2
            (fn () => passive_sync belt2_ready_for_crane),
            passive_sync_event belt2_ready_for_crane
            (fn () => passive_sync belt2_ready_for_arm2)];
        belt2_cycle ()
    in spawn(fn () => (passive_sync belt2_ready_for_arm2;
                         belt2_cycle ())) end;
fun move_until wait_fun start_ch stop_ch =
  ( send(start_ch,())
    ; wait_fun ()
    ; send(stop_ch,()) )

is assigned the annotated type

\[(\text{unit} \rightarrow^\beta_0 \alpha_0) \rightarrow^\beta_1 \text{unit} \text{ chan } \rho_0 \rightarrow^\beta_1 \text{ unit} \text{ chan } \rho_1 \rightarrow^\beta_2 \text{ unit}\]

with the “meaning” of $\beta_1$ and $\beta_2$ given by

$$
\varepsilon \subseteq \beta_1 \\
(\rho_0!\text{unit});\beta_0;(\rho_1!\text{unit}) \subseteq \beta_2
$$
--- Behaviours: syntax

\[
\begin{align*}
b &::= \quad ae | \varepsilon | b_1; b_2 | b_1 + b_2 | \beta \\
\end{align*}
\]

Sequential composition ";" associative, with \( \varepsilon \) as neutral element

Atomic effects:

\[
\begin{align*}
ae &::= \quad \text{SPAWN } b \mid t \text{ CHAN } r \\
&\quad | \quad r!t \mid r?t \\
\end{align*}
\]

Types:

\[
\begin{align*}
t &::= \quad \alpha | \text{unit} | \text{bool} | \text{int} | t_1 \rightarrow^\beta t_2 \\
&\quad | \quad t_1 \times t_2 | t \text{ list} | t \text{ chan } r \mid t \text{ event } \beta \\
\end{align*}
\]

Regions:

\[
\begin{align*}
r &::= \rho | \{l\} | r_1 \cup r_2 \\
\end{align*}
\]
— Methodology for validating protocols

Semantic soundness: well-typed programs

- do not go wrong
- behave according to their behaviour (i.e., behaviour simulates program)

Suppose a program $P$ has behaviour

$$ae_1; (ae_2 + ae_3)$$

These runtime actions are possible:

- first $ae_1$ then $ae_2$
- first $ae_1$ then $ae_3$
- first $ae_1$ then block
- first $ae_1$ then loop (actually not!)
- block or loop initially

These runtime action are impossible:

- first $ae_2$ (if $ae_1 \neq ae_2$)
- ...
Inference system

Judgements take the form \( C, A \vdash e : t \& b \)
with \( C \) constraint set and \( A \) environment

[id] \( C, A \vdash x : A(x) \& \varepsilon \)

[abs] \[
\frac{C, A[x : t_1] \vdash e : t_2 \& \beta}{C, A \vdash \text{fn } x \Rightarrow e : t_1 \to^\beta t_2 \& \varepsilon}
\]

[app] \[
\frac{C, A \vdash e_1 : t_2 \to^\beta t_1 \& b_1 \quad C, A \vdash e_2 : t_2 \& b_2}{C, A \vdash e_1 \; e_2 : t_1 \& (b_1; b_2; \beta)}
\]

\[
\frac{C, A \vdash e_0 : \text{bool} \& b_0 \quad C, A \vdash e_1 : t \& b_1 \quad C, A \vdash e_2 : t \& b_2}{C, A \vdash \text{if } e_0 \text{ then } e_1 \text{ else } e_2 : t \& b_0; (b_1 + b_2)}
\]
Subtyping

\[ C, A \vdash e : t \& b \quad \text{if} \quad C \vdash \{ t \subseteq t', b \subseteq b' \} \]

\[ C, A \vdash e : t' \& b' \]

\( C \vdash b \subseteq b' \) means that (modulo \( C \))

\( b' \) less precise than \( b \), similarly for types

Selected rules:

\[ C \vdash b \subseteq \beta \quad \text{if} \quad (b \subseteq \beta) \in C \]

\[ C \vdash b_1 \subseteq b_1 + b_2 \]

\[ C \vdash b_1 \subseteq b \quad C \vdash b_2 \subseteq b \]

\[ C \vdash b_1 + b_2 \subseteq b \]

\[ C \vdash t'_1 \subseteq t_1 \quad C \vdash \beta \subseteq \beta' \quad C \vdash t_2 \subseteq t'_2 \]

\[ C \vdash t_1 \rightarrow^\beta t_2 \subseteq t'_1 \rightarrow^\beta t'_2 \]

\( t \) chan \( r \) co- as well as contravariant in \( t \)

Property: if \( \emptyset \vdash t_1 \subseteq t_2 \)

then \( t_1 \) and \( t_2 \) have same shape
(e.g., \( t_1 = \text{int} \) implies \( t_2 = \text{int} \))
Polymorphism

if \( C \cup C_0, A[f : t_1 \rightarrow^{b_0} t_2][x : t_1] \vdash e_0 : t_2 \& b_0 \)

and \( C, A[f : \forall(\alpha\beta\rho : C_0). t_1 \rightarrow^{b_0} t_2] \vdash e : t \& b \)

then \( C, A \vdash \text{let fun } f \ x = e_0 \text{ in } e \text{ end} : t \& b \)

provided \( \cdots \)

To type \( \text{let val } x = e_0 \text{ in } e \text{ end} \) with \( e_0 \) not a function, even hairier
Inference algorithm

Goal: define $\mathcal{W}$ such that

$$\mathcal{W}(A, e) = (S, t, b, C) \text{ implies } C, S A \vdash e : t \& b$$

$\mathcal{W}(A, \text{fn } x \Rightarrow e_0) =$

let $\alpha$ be fresh
let $(S_0, t_0, b_0, C_0) = \mathcal{W}(A[x : \alpha], e_0)$
let $\beta$ be fresh
in $(S_0, S_0 \alpha \rightarrow^{\beta} t_0, \varepsilon, C_0 \cup \{b_0 \subseteq \beta\})$

$\mathcal{W}(A, \text{channel}^l) =$

let $\alpha$, $\beta$, $\rho$ be fresh
in $(\text{Id, unit} \rightarrow^{\beta} \alpha \text{ chan} \rho, \varepsilon, \{\alpha \text{ CHAN } \rho \subseteq \beta, \{l\} \subseteq \rho\})$
— Generation of atomic constraints —

If branches of a conditional have been given types

\[
\text{int} \rightarrow^{\beta_1} \alpha_1 \quad \text{and} \quad \text{int} \rightarrow^{\beta_2} \alpha_2
\]

we lose precision if we unify \(\beta_1\) with \(\beta_2\) or \(\alpha_1\) with \(\alpha_2\)

So instead we generate constraints

\[
\beta_1 \subseteq \beta, \ \beta_2 \subseteq \beta, \ \alpha_1 \subseteq \alpha, \ \alpha_2 \subseteq \alpha
\]
 Decomposing constraints

\[ W(A, e_1 e_2) = \]
\[ \text{let } (S_1, t_1, b_1, C_1) = W(A, e_1) \]
\[ \text{let } (S_2, t_2, b_2, C_2) = W(S_1 A, e_2) \]
\[ \text{let } \alpha, \beta \text{ be fresh} \]
\[ \text{in } (S_2 S_1, \alpha, (S_2 b_1; b_2; \beta), S_2 C_1 \cup C_2 \cup \{S_2 t_1 \subseteq t_2 \rightarrow^\beta \alpha\}) \]

Need to decompose constraints into atomic:
(to ensure termination, some extra device is needed)

\[ \{t_1 \rightarrow^\beta t_2 \subseteq t'_1 \rightarrow^\beta' t'_2\} \rightarrow \{t'_1 \subseteq t_1, \beta \subseteq \beta', t_2 \subseteq t'_2\} \]
\[ \{t_1 \rightarrow^\beta t_2 \subseteq \text{int}\} \rightarrow \text{fail} \]
\[ \{\alpha \subseteq t_1 \rightarrow^\beta t_2\} \rightarrow \{\alpha_1 \rightarrow^\beta' \alpha_2 \subseteq t_1 \rightarrow^\beta t_2\} \]
recording substitution with \(\alpha_1, \alpha_2, \beta'\) fresh
Simplifying constraints

Awful lot of atomic constraints!

To greatly improve performance, without losing precision, remove variables as suggested below:

Suppose that $\alpha_1$ is the “greatest lower bound” of $\alpha_2$.

We can then shrink $\alpha_2$ into $\alpha_1$ globally, if $\alpha_2$ occurs positively in the type $t$ as is the case for $t = \alpha_1 \rightarrow \alpha_2$.

Intuitive justification: the old type

$$(\alpha_1 \rightarrow \alpha_2 \text{ wrt. } \{\alpha_1 \subseteq \alpha_2\})$$

and the new type

$$\alpha_1 \rightarrow \alpha_1 \text{ wrt. } \emptyset$$

have the same “information content”:

$$\{t_1 \rightarrow t_2 \mid t_1 \subseteq t_2\}$$
Post-processing constraints

Still awful lot of constraints
(deposit belt: 200+ for behaviours alone)

Suppose \( \mathcal{W} = (\_, \_, b, C) \) with

\[
    b = \beta_0; \beta_1; (\text{SPAWN } \beta_2); \beta_3
\]

\[
    C = \{(\{0\} \subseteq \rho_0), (\{1\} \subseteq \rho_1), (\rho_1 \subseteq \rho_2),
        (\text{unit CHAN } \rho_0 \subseteq \beta_0), (\text{unit CHAN } \rho_1 \subseteq \beta_1),
        (\rho_2 !\text{unit}; \rho_0 ?\text{unit} \subseteq \beta_2),
        (\rho_2 ?\text{unit}; \rho_0 !\text{unit} \subseteq \beta_3)\}
\]

If say channel label 1 is not of interest, we get by “unfolding” the behaviour

\[
    \text{unit CHAN } \{0\}; \tau; \text{SPAWN } (\tau; \{0\} ?\text{unit}); \tau; \{0\} !\text{unit}
\]

“Code sharing” useful: if \( \beta_1, \beta_2 \) given by

\[
    (\{7\} !\text{int}; \beta_1 \subseteq \beta_1) \text{ and } (\{7\} !\text{int}; \beta_2 \subseteq \beta_2)
\]

then \( \beta_1 \) is globally replaced by \( \beta_2 \)

Correctness formulated as bisimulation:

\( (b, C) \) transformed to \( (b', C') \sim (b, C) \)
Region representation

If program contains

let val ch7 = channel⁴ () in ... end
then we write \{ch7\} instead of \{4\}.

Caveat wrt. polarity:

let fun f ch = if ... then ch
    else channel⁰ ()
in f end

\(\mathcal{W}\) returns

- type \(\alpha \text{ chan } \rho \rightarrow^{\beta} \alpha \text{ chan } \rho\)
- constraint set including (\(\{0\} \subseteq \rho\))

Least solution: map \(\rho\) into \(\{0\}\)

Desired solution: map \(\rho\) into \(\{0\} \cup r_\rho\)
Analyzing the deposit belt

Interpretation of inferred behaviour:

unit chan {belt2_start}; unit chan {belt2_stop};
a0 chan {belt2_blank_at_end}; a1 chan {belt2_no_blank_at_end};
unit chan {belt2_ready_for_arm2};
unit chan {belt2_ready_for_crane}; spawn (B0; B1)

Interpretation of type/behaviour variables:

B0 ‘means’ {belt2_ready_for_arm2}!unit;
    {belt2_ready_for_arm2}?unit
B1 ‘means’ {belt2_start}!unit; {belt2_blank_at_end}?a0;
    {belt2_no_blank_at_end}?a1; {belt2_stop}!unit;
    ({belt2_ready_for_crane}!unit;
    {belt2_ready_for_crane}?unit; B0
    + {belt2_ready_for_arm2}!unit;
    {belt2_ready_for_arm2}?unit;
    {belt2_ready_for_crane}!unit;
    {belt2_ready_for_crane}?unit); B1
Focusing on `belt2_start`:

**Interpretation of inferred behaviour:**

```
unit chan (belt2_start); tau_5; fork (tau_2; B_0)
```

**Interpretation of type/behaviour variables:**

```
B_0 'means' (belt2_start)! unit; tau_7; B_0
```

**Type schemes of let-bound identifiers:**

```
move_until
passive_sync
passive_sync_event
belt2
belt2_cycle
```
Case study: validating software

We are given a CML program for the production cell

- developed using systematic design methods
- functionality specified in CSP
- many safety conditions formally verified

Let’s test it!

Protocols for communication between system parts somewhat different from those used in our programs
Testing for blank collisions

No blanks must be put on top of each other on the deposit belt

Channels of interest:

- `belt2_start`: a request starts the belt
- `arm2_transmit_ready`: a request signals that the robot drops a blank on the belt
— Output from prototype —

Interpretation of inferred behaviour:
...;unit chan {belt2_start};...;unit chan {arm2_transmit_ready};
...;B0;B0;B0;spawn B1;spawn B2;B0;spawn B3

Interpretation of type/behaviour variables:
B0 'means' spawn ...
B1 'means' ...;B1
B2 'means' {arm2_transmit_ready}?unit;...;B4;B5
B3 'means' ...;B6
B4 'means' {belt2_start}!unit;...
B5 'means' ...;B2 + {arm2_transmit_ready}?unit;...;B4;B5
B6 'means' ...;B7
B7 'means' ...
  (...;B8
   + ...;(...;B8 + ...;{arm2_transmit_ready}!unit;...);
   B3)
B8 'means' ...
  ({arm2_transmit_ready}!unit;...;B6
   + ...;{arm2_transmit_ready}!unit;...;B7)
--- Massaging the output  ________

B2 is the behaviour of the belt.

B2 'means' {arm2_transmit_ready}?unit;...;
    {belt2_start}!unit;...;B5
B5 'means' ...;B2 +
    {arm2_transmit_ready}?unit;...;
    {belt2_start}!unit;...;B5

We have validated: no blank collisions!
Testing if table obeys vertical bounds

Table must not be moved upwards if in upper position;
table must not be moved downwards if in lower position

Channels of interest:

- \texttt{table\_upward}, \texttt{table\_downward}: request is the way to initiate table movement
- \texttt{table\_is\_top}, \texttt{table\_is\_bottom}, \texttt{table\_is\_not\_top}, \texttt{table\_is\_not\_bottom}: “sensor” channels
-- Output from prototype, revealing initialization error .

Interpretation of inferred behaviour:

...;unit chan {table_upward};...;unit chan {table_downward};...
a0 chan {table_is_bottom};a1 chan {table_is_not_bottom};
a2 chan {table_is_top};a3 chan {table_is_not_top};...;B0;B0;B0;
spawn B1;spawn B2;spawn B3;spawn B4

Interpretation of type/behaviour variables:

B0 'means' spawn ...
B1 'means' ...;B1
B2 'means' ...;B5
B3 'means' ...;{table_upward}!unit;{table_is_top}?a2;...
   {table_downward}!unit;{table_is_bottom}?a0;...;B3
B4 'means' ...;B6
B5 'means' ...;B2 + ...;B5
B6 'means' ...;B7
B7 'means' ...;(...;B8 + ...;(... + ...;B8);B4)
B8 'means' ...;(...;B6 + ...;B7)
— Testing if table obeys horizontal bounds

Table must alternate between clockwise and counterclockwise rotation

Channels of interest:

- `table_left`, `table_right`, `table_stop_h`: requests control table movements
- `table_angle`: sensor channel
...;unit chan {table_left};unit chan {table_stop_h};
unit chan {table_right};...;int chan {table_angle};...;B0;B0;B0;
spawn B1;spawn B2;spawn B3;spawn B4

B0 'means' spawn ...
B1 'means' ...;B1
B2 'means' ...;B5
B3 'means' ...;B6;...;B6;B3
B4 'means' ...;B7
B5 'means' ...;B2 + ...;B5
B6 'means' {table_angle}?int;
  (e
    + {table_left}?unit;B8;{table_stop_h}?unit
    + {table_right}?unit;B8;{table_stop_h}?unit)
B7 'means' ...;B9
B8 'means' ...;{table_angle}?int;(e + B8)
B9 'means' ...;(...;B10 + ...;(... + ...;B10);B4)
B10 'means' ...;(...;B7 + ...;B9)
A source program and its intermediate form

Let \( \text{val chA} = \text{channel}() \) \( \text{val chB} = \text{channel}() \)
\( \text{val chC} = \text{channel}() \)
\( \text{val chD} = \text{chB} \)
\( \text{val chE} = \text{if true then chA else channel}() \)
\( \text{in} \ (\text{chA}, \text{chB}, \text{chC}, \text{chD}, \text{chE}) \) end

Translation to intermediate language:

\[
\begin{align*}
&\text{let chA} = (\text{channel}^0()) \text{ in} \\
&\text{let chB} = (\text{channel}^1()) \text{ in} \\
&\text{let chC} = (\text{channel}^2()) \text{ in} \\
&\text{let chD} = \text{chB in} \\
&\text{let chE} = (\text{if true then chA else (channel}^3())()) \\
&\text{in} ((\text{pair chA}) \\
&\quad ((\text{pair chB}) ((\text{pair chC}) ((\text{pair chD}) \text{chE}))))
\end{align*}
\]
The relation between program identifiers and channel labels

```plaintext
let val chA = channel () val chB = channel ()
  val chC = channel ()
  val chD = chB
  val chE = if true then chA else channel ()
in (chA, chB, chC, chD, chE) end
```

Type schemes of let-bound identifiers:

<table>
<thead>
<tr>
<th>Label</th>
<th>Will be bound to</th>
<th>May be bound to</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>chA</td>
<td>chE</td>
</tr>
<tr>
<td>1</td>
<td>chB, chD</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>chC</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>chE</td>
</tr>
</tbody>
</table>
Displaying type and behaviour information

Interpretation of inferred type:

\[ a_0 \text{ chan } \{0\} \times (a_1 \text{ chan } \{1\} \times (a_2 \text{ chan } \{\text{chC}\} \times (a_1 \text{ chan } \{1\} \times a_0 \text{ chan } \{0,3\}\}))) \]

Interpretation of inferred behaviour:

\[ a_0 \text{ chan } \{0\}; a_1 \text{ chan } \{1\}; a_2 \text{ chan } \{\text{chC}\}; (e + a_0 \text{ chan } \{3\}) \]

Time spent in ‘decode’:

- real time: 6 ms.
- user cpu time: 6 ms including go time: 0 ms. sys time: 0
— Conclusion —

Developed a tool for validating software

- firm semantic foundation
- complementary to formal development
- thus helps to keep the trusted base small
- handles higher-order and polymorphic programs
- but still very sensitive to programming style

Documentation:

- Amtoft, Nielson & Nielson (Imperial College Press, 1999): *Type and Effect Systems: Behaviours for Concurrency*