SMAWL

A SMAll Workflow Language Based on π-Calculus [1]

Christian Stefansen `cstef@diku.dk`

SMAWL is a workflow language based on π-Calculus[1] that:

- can express all 20 workflow patterns identified by Aalst/Hofstede [2] and more
- can be used both graphically and textually
- maintains strong ties with its theoretical foundation through a source-level translation to CCS (Calculus of Communicating Systems)
- enables immediate use of current verification tools (simulation tools, model checkers, etc.)
- is readable and high-level

In essence, SMAWL reaps the benefits of (1) graphical representation, (2) textual representation, and (3) low-level representation for automation/verification

Low-Level Manipulation

$\mathcal{T}[\![\cdot]\!] : SMAWL \rightarrow Channel \rightarrow CCS$ recursively maps SMAWL expressions to CCS expressions. Here are some examples of how this is done (see [1] for the complete transformations):

$$\mathcal{T}[\![P;Q]\!] = \lambda ok.\text{let } ok' <= \nu() \text{ in } \mathcal{T}[\![P]\!]ok' \mid ok'?.\mathcal{T}[\![Q]\!]ok$$

$$\mathcal{T}[\![\textbf{choose one } \{=> P_1 => \cdots => P_n\}]\!] = \lambda ok.\mathcal{T}[\![P_1]\!]ok + \cdots + \mathcal{T}[\![P_n]\!]ok$$

The generated expression can then be model checked, simulated, etc. in existing tools.

$\mathcal{T}[\![\cdot]\!]$ $\longleftarrow$

verification $\longrightarrow$

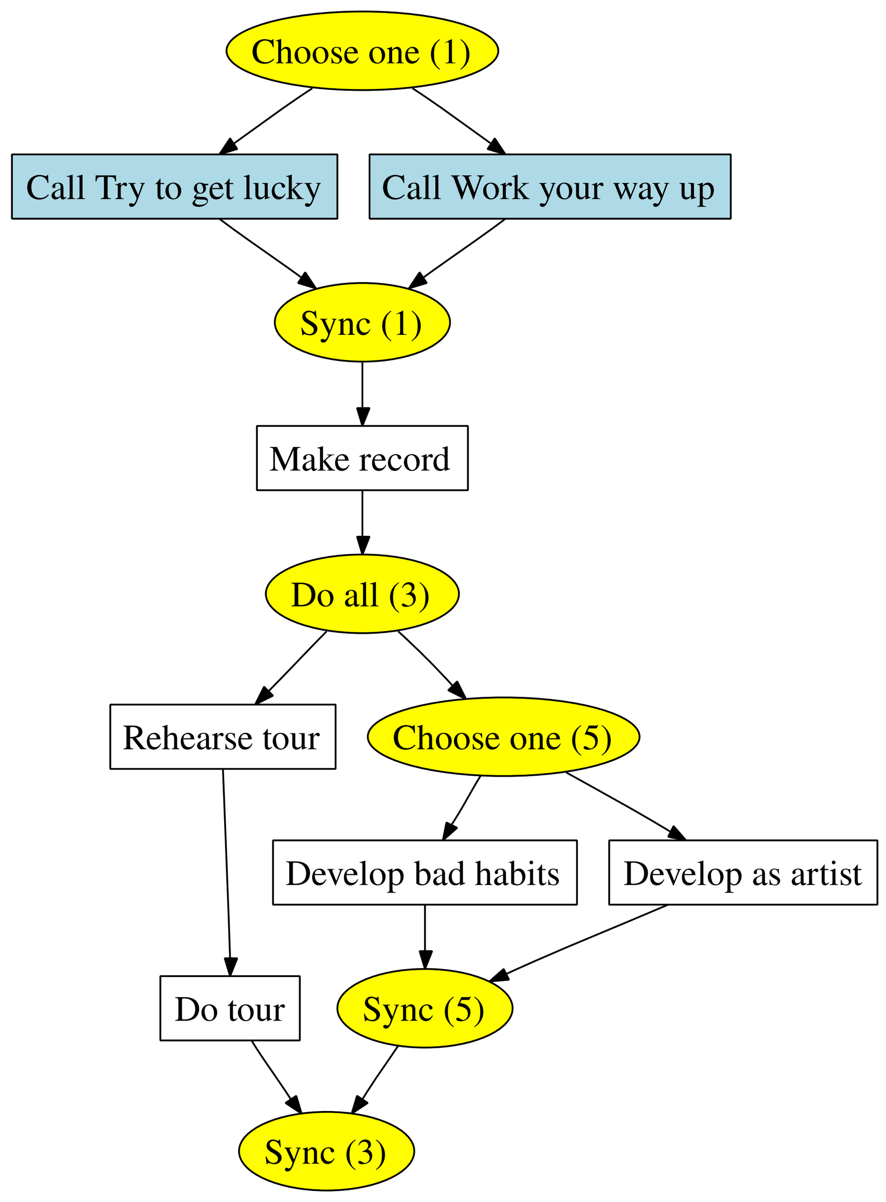
Textual Manipulation

workflow *Become a recording star* =
  chooseone {
    => **call** (*Work your way up*)
    => **call** (*Try to get lucky*)
  };
  Make record;
  doall {
    => **chooseone** {
      => *Develop as an artist*
      => *Develop bad habits*
      }
    => *Rehearse tour*;
      Do tour
  }
end

autogenerated $\longrightarrow$

autogenerated* $\longleftarrow$

Graphical Manipulation



Syntax and Description

SMAWL is parameterized over the dataflow/predicate language. This important property makes SMAWL plugable with a wide range of languages and keeps it compact:

$$\begin{array}{rcl}
Prog & ::= & DD\ \textbf{workflow}\ w = P\ \textbf{end} \\
DD & ::= & \textbf{fun}\ f = P\ \textbf{end}\ DD \mid \textbf{newlock}\ (l,u)\ DD \\
 & \mid & \textbf{milestone}(ison, isoff, set, clear)\ DD \mid \epsilon \\
P & ::= & activity \mid \textbf{send}(f) \mid \textbf{receive}(f) \mid \textbf{call}(f) \mid P\textbf{;}\ P \mid \textbf{lock}(l,u)\{P\} \\
 & \mid & \textbf{choose any (wait for } k\textbf{)}\{PP\ \textbf{merge}(n)\ P\} \mid \textbf{choose one}\{PP\} \\
 & \mid & \textbf{do all (wait for } k\textbf{)}\{PP\ \textbf{merge}(n)\ P\} \mid \textbf{multi}(n)\{P\} \mid \textbf{cancel}\ \{P\} \\
PP & ::= & => \rho\ P\ PP \mid => P\ PP \mid \epsilon
\end{array}$$

(Square brackets denote the workflow patterns covered by each construct.)

$activity$ indicates an atomic activity to be carried out.

P**;** Q is the sequence pattern waiting for P to finish before starting Q. [Sequence]

choose one$\{PP\}$ does exactly one of the processes in the list PP. Each of processes in the list can be guarded with a predicate ρ or not and hence this construct can express both deferred choice, explicit choice, and any combination thereof. [Exclusive Choice, Deferred Choice, Simple Merge]

choose any (wait for k**)**$\{PP$ **merge**(n) $Q\}$ does any number of the processes in the list PP, spawns the process Q for the first n to finish, and continues once k instances of Q have finished. [Multiple Choice, Deferred Multiple Choice, Multiple Merge, N-out-of-M Join, Synchronizing Merge, Discriminator]

do all (wait for k**)**$\{PP$ **merge**(n) $Q\}$ starts all processes in the list PP, spawns the process Q for the first n to finish, and continues once k instances of Q have finished.[Parallel Split, Synchronization, Multiple Merge, N-out-of-M Join, Synchronizing Merge, Discriminator]

multi$(n)\{P\}$Starts multiple instances of the process P. Execution continues once all spawned processes are done – i.e. synchronization is performed [MI with a priori known design time knowledge, MI with/without a priori known runtime knowledge].

fun $f = P$ **end** declares a sub-workflow callable using **call**(f). **call**(f) calls a declared sub-workflow f and blocks until it finishes. [MI without synchronization]

send(f)/**receive**(f) provide blocking primitives for signals to locks, milestones, arbitrary joins, and cancellable processes. They are the send and receive primitives found in CCS. [Arbitrary Cycles]

newlock (l,u) declares a new global lock. **lock**$(l,u)\{P\}$protects process P through the declared lock (l,u). [Interleaved Parallel Routing]

milestone$(ison, isoff, set, clear)$ declares a milestone that can be read/set by any process knowing the correct channels. [Milestone]

cancel $\{P\}$makes the process P cancellable on a pre-determined signal c. The property does not penetrate functions unless specifically stated in their definition. [Cancel Activity, Cancel Case]

References

[1] Christian Stefansen. SMAWL: A SMAll Workflow Language based on CCS. Technical Report TR-06-05, Harvard University, Division of Engineering and Applied Sciences, Cambridge, MA 02138, March 2005.

[2] W.M.P. van der Aalst and A.H.M. ter Hofstede. Workflow patterns: On the expressive power of (petri-net-based) workflow languages. In K. Jensen, editor, *Proceedings of the Fourth Workshop on the Practical Use of Coloured Petri Nets and CPN Tools (CPN 2002)*, volume 560, Aarhus, Denmark, August 2002. DAIMI.

Also see our website at `http://topps.diku.dk/next/`.

Microsoft Business Solutions

DET NATURVIDENSKABELIGE FAKULTET
KØBENHAVNS UNIVERSITET

DIKU

IT University of Copenhagen

[1] Actually, it is even better than that: the language is based entirely on CCS (Calculus of Communicating Systems), the simpler subset of π-calculus that does not allow channel-passing.