SMAWL: A SMAll Workflow Language Based on CCS

Christian Stefansen

Department of Computer Science, University of Copenhagen (DIKU)
Universitetsparken 1, 2100 Copenhagen Ø, Denmark

Abstract This paper provides a overview of SMAWL, a SMAll Workflow Language based on CCS (Calculus of Communicating Systems). There has been a prolonged debate in the workflow community about the relative suitability of Petri nets versus π-calculus as a formal foundation for workflow languages. Here we demonstrate how to build a workflow language based on CCS (a predecessor of π-calculus). To facilitate comparison with other approaches SMAWL is designed to be able to express the same 20 patterns that originally led to the design of the Petri net-based workflow language YAWL by van der Aalst and ter Hofstede. After an initial example of a SMAWL program, some design considerata are discussed, and the constructs of the language are presented along with excerpts of the compositional source-level translation to CCS.

1 Introduction

There has been a long debate in the workflow community about the relative merits of different formalisms – most notably π-calculus [2] and Petri nets – for workflow modeling [5,8]. Proponents of the π-calculus claim that the presence of mobility or, more specifically, channel-passing, makes it the more suitable choice. Proponents of Petri nets have pointed out that contrary to π-calculus, Petri nets have a standardized and rigorous graphical notation readily available.

In [6] van der Aalst and ter Hofstede identified 20 common workflow patterns, surveyed current workflow systems with respect to these, and presented a Petri net-based workflow language, YAWL, that is capable of expressing all 20 patterns. In the same vein SMAWL is designed with this benchmark in mind, but based on CCS. Since it is always desirable to keep the formal foundation as simple as possible, and since the language design did not require the notion of mobility found in π-calculus, SMAWL is based on CCS (Calculus of Communicating Systems) rather than π-calculus. Here "based on CCS" means translatable to CCS using only source-code transformations.

The 20 workflow patterns do not deal with data flow. This makes the patterns easier to work with and forces a strong separation between control flow and data flow. By following this strategy and parameterizing over the data-flow language, SMAWL avoids being tied in, but may be used with any (sensible) data-flow language.

1.1 Why Consider CCS?

CCS and π-calculus have been studied extensively for years and have sound mathematical foundations. There has been a wealth of practical applications in programming

languages (e.g. PICT [3]), protocol verification, software model checking (e.g. [1]), hardware design languages, and several other areas.

Workflow languages seem to have a lot in common with these areas: they can be thought of as parallel processes and often defy the block structure found in conventional programming. It seems appropriate to leverage the strong separation of process and application logic enforced in CCS, and furthermore, having a well-understood foundation is likely to make implementation and subsequent adaptation easier and more routine. By using CCS we can integrate with existing tools for automated verification and thus make writing and adapting workflows less error-prone.

Nevertheless, CCS presents significant challenges. Workflow systems should be accessible to anyone, and so a central challenge is to provide graphical tools and user-friendly abstractions. It is important to note that while not inherently graphical, CCS does not preclude the use of a high-level graphical representation. We do not mean to use CCS directly as it is, but as a theoretical foundation for future work. This paper suggests how.

1.2 Contributions

The main contribution of the paper is an overview of the language SMAWL: a CCS-based compositional workflow language that can express all 20 patterns identified in [6]. The language deals with most of the internal message-passing required to code the 20 workflow patterns and provides a palatable syntax for programmers. Specifically, we (1) describe the grammar and each of the constructs, (2) outline the formal translation to CCS, (3) present a small example, and (4) show the auto-generated graphical representation hereof to suggest that graphical manipulation can be implemented in a relatively simple way. Interested readers are referred to the accompanying technical report [4] for a more detailed walkthrough.

1.3 Outline

Section 2 shows an example workflow, describes the constructs and design deciderata of SMAWL, and sketches its formal translation into CCS. Section 3 contains a few concluding remarks.

2 Introducing SMAWL

The goal is to design a CCS-based language that (a) reduces the amount of user-specified internal synchronization required and (b) provides elegant constructs for the 20 workflow patterns [6] shown in Table 1.

To get a feel for where this will lead, an example workflow in the resulting language can be seen in Figure 1. The main workflow, which hopefully is self-explanatory, specifies that to become a recording artist one should first either “Work your way up” or “Try to get lucky”. After the chosen subroutine is done, one should “Make record” and finally develop a personality according to one’s own choice in parallel to first “Rehearse tour” and then “Do tour”.

Basic Control Patterns

1 Sequence
2 Parallel Split
3 Synchronization
4 Exclusive Choice
5 Simple Merge

Advanced Branching and Synchronization Patterns

6 Multiple Choice
7 Synchronizing Merge
8 Multiple Merge
8a N-out-of-M Merge *(new)*
9 Discriminator
9a N-out-of-M Join

Patterns Involving Multiple Instances

12 MI without synchronization
13 MI with a priori known design time knowledge
14 MI with a priori known runtime knowledge
15 MI with no a priori runtime knowledge

Structural Patterns

10 Arbitrary Cycles
11 Implicit Termination

Cancellation Patterns

19 Cancel Activity
20 Cancel Case

State-Based Patterns

16 Deferred Choice
16a Deferred Multiple Choice *(new)*
17 Interleaved Parallel Routing
18 Milestone

Table1. The 20 Workflow Patterns [7] and Two New Ones

The graphical representation was generated completely automatically from the abstract syntax tree of the code to give an example that even with CCS as the foundation it is relatively straight forward to obtain an intuitive user interface.

2.1 Designing SMAWL

Expressing the patterns directly in CCS is a cumbersome and error-prone exercise in low-level coding. In particular the programmer must write the specifics of synchronization every time a merge or join pattern is used. It is therefore a natural idea to make small building blocks – combinators – of each of the patterns. This way there will be e.g. a number of split combinators and a number of join combinators etc.

However, it turns out to be very tedious for the programmer to explicitly synchronize every time a split block is left; a more palatable approach is therefore to implicitly synchronize after every split construct and have the programmer explicitly write if the continuation should be spawned for each active thread (a merge). These and numerous other considerations lead to the following syntax:

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

```
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
```

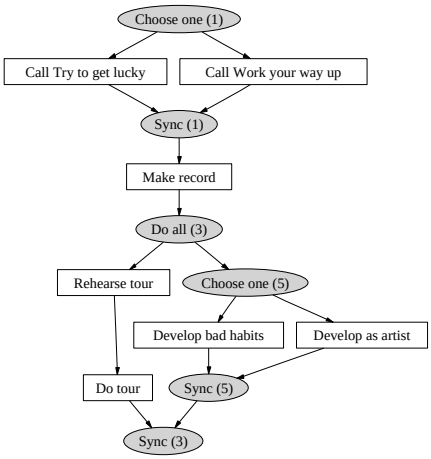

Figure1. How To Become a Recording Star (adapted from the *Recording Star* example *[7])*

In the syntax ϵ denotes the empty string, f is a function name, w is the workflow name, ρ is a data-dependent predicate [1], k is a natural number and n is a natural number or ∞. ρ is what allows data-dependent choices, all other choices default to deferred choices.

A program is any number of declarations DD followed by a named main workflow process P. Let us informally consider each of the constructs in turn indicating the patterns that they cover in square brackets:

$activity$ 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. Through the predicate ρ it also provides part of the interface to the dataflow language. [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. More technically **choose any (wait for** k**)**$\{PP$ **merge**(n) $Q\}$ implements multiple choice over the PPs, then merges each of the threads to Q, and finally synchronizes all threads. If the clause **wait for** k is given, the synchronization will be an *N-out-of-M Join*. If the clause is not provided, synchronization will wait for all threads to signal done. In the **merge** part of the clause a value of $n = \infty$ signifies all threads PP should merge to Q upon completion. A value of $n \neq \infty$ signifies that only the first n threads to complete should give rise to an instantiation of Q. If **merge**(n) Q is missing, n is taken to be ∞ and $Q = \mathbf{0}$. [Multiple Choice, Deferred Multiple Choice, Multiple Merge, N-out-of-M Join, Synchronizing Merge, Discriminator]

[1] The format of predicates ρ is not of the essence here; such predicates will simply be compiled to a τ prefix and the responsibility of deciding them will be delegated to a data-aware layer.

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. It accepts the same options as **choose any (wait for** k**)**$\{PP$ **merge**$(n)\ Q\}$ for merging and synchronizing. [Parallel Split, Synchronization, Multiple Merge, N-out-of-M Join, Synchronizing Merge, Discriminator]

multi$(n)\{P\}$Starts multiple instances of the process P. If n is a natural number then exactly that number of copies will be spawned. If $n = \infty$ then processes P can emit a designated signal to spawn more instances while running or more instances can be spawned based on a data-layer predicate. 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. Global meaning that two processes that are not allowed to run at the same time, do not have to be within the same logical block. **lock**$(l, u)\{P\}$protect process P through the declared lock (l, u). Signals lock on l and unlock on u on entering/exiting the process P. [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]

2.2 Compiling SMAWL to CCS

Compiling the workflow description language is a relatively simple task since the language is based directly on patterns that have already been described in CCS (see the accompanying technical report [4]). The main transformation $\mathcal{T}[\![\cdot]\!]$ is a map $Prog \rightarrow Channel \rightarrow CCS$, where $Channel$ denotes the set of valid channel names. So given a SMAWL program and a channel name c, the transformation $\mathcal{T}[\![Prog]\!]c$ will generate a CCS expression that signals on c when the workflow has finished. The CCS expressions are formed using the following syntax:

$$P ::= \mathbf{0} \mid \tau.P \mid a?x.P \mid a!x.P \mid P + P \mid (P \mid P) \mid \mathsf{new}\ a\ P \mid a?*x.P$$

where $a?*x.P$ spawn the process P each time a message is received on a. The remaining operators are standard.

Consider the transformation of the sequence P**;** Q. When done, P should signal to Q to continue, and Q when done should signal to the outside world on a designated channel. A helper function is needed: $\nu : \{()\} \rightarrow Channel$ returns a fresh channel name that has not previously been used. Now P and Q can communicate, and the transformation becomes:

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

Now consider the more complex pattern defined by:

$$\begin{aligned}\mathcal{T}[\![\textbf{do all (wait for k) } \{\Rightarrow P_1 \Rightarrow \cdots \Rightarrow P_n \ (\textbf{merge } l\ Q)\}]\!] = \\ \lambda ok.\text{let } ok' \Leftarrow \nu() \text{ in let } ok'' \Leftarrow \nu() \text{ in let } ok''' \Leftarrow \nu() \text{ in} \\ \mathcal{T}[\![P_1]\!]ok' \mid \cdots \mid \mathcal{T}[\![P_n]\!]ok' \mid \text{mergeprefix}(l, ok', ok'') \\ \mid ok''?*.\mathcal{T}[\![Q]\!]ok''' \mid \underbrace{ok'''?.\cdots.ok'''?}_{k}.(ok! \mid ok'''?*)\end{aligned}$$

where mergeprefix is the map $\mathbb{N} \cup \{\infty\} \times Channel \times Channel \rightarrow CCS$ defined by:

$$\begin{aligned}\text{mergeprefix}(\infty, ok, go) &= ok?*.go! \\ \text{mergeprefix}(n, ok, go) &= \underbrace{ok?.go!.\cdots.ok?.go!}_{n}.ok?*\end{aligned}$$

Interested readers should consult the accompanying technical report [4] for the complete transformations. Interestingly, it turns out new operators can be statically removed by alpha conversion – bar the rare cases where new is used inside replicated processes.

3 Conclusion

We presented a language that makes CCS-based workflow systems more accessible. SMAWL turned out to be an easy and very convenient language for writing workflow expressions and more work will conducted in this direction. The language SMAWL is kept very simple and yet it is powerful enough to express the workflow patterns we have seen to far. It would be interesting to see how far one can get in terms of graphical support for SMAWL, and equally it would be interesting to plug SMAWL into a formal verification tool.

References

1. Tony Andrews, Shaz Qadeer, Sriram K. Rajamani, and Yichen Xie. Zing: Exploiting program structure for model checking concurrent software. In *CONCUR*, 2004.
2. Robin Milner. *Communicating and Mobile Systems: The π-calculus*. Cambridge University Press, 1999.
3. Benjamin C. Pierce and David N. Turner. Pict: A programming language based on the pi-calculus. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 2000.
4. 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.
5. W.M.P. van der Aalst. Pi calculus versus Petri nets: Let us eat "humble pie" rather than further inflate the "Pi hype". Available from http://tmitwww.tm.tue.nl/research/patterns/download/pi-hype.pdf, 2004.
6. 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.
7. Workflow patterns. Available from http://www.workflowpatterns.com.
8. Michael zur Muehlen. Workflow research. http://www.workflow-research.de.