SMAWL: A SMAll Workflow Language Based on CCS

Christian Stefansen

Division of Engineering and Applied Sciences
Harvard University

Abstract. There is an ongoing debate in the workflow community about the relative merits of Petri nets and π-calculus for workflow modeling. Recently, van der Aalst presented some challenges to model workflow in π-calculus. This paper responds to those challenges by showing how to code the 20 most commonplace workflow patterns in CCS (a subset of π-calculus), and describes two new workflow patterns that were identified in the process. It then presents a CCS-based language, SMAWL, as a result of this work and shows how the *Recording Star* example can be expressed in SMAWL. The applicability of π-calculus to the workflow modeling domain is briefly discussed and a new *overlaying* operator is discussed with applications to workflow descriptions.

1 Introduction

A prolonged debate has been going on about the relative merits of Petri nets and π-calculus for workflow modeling [zM]. Nonetheless, very little of this debate has been published or put in writing elsewhere. Recently van der Aalst presented seven challenges for those argueing in favor of π-calculus as a formal foundation for workflow systems [vdA04]. Challenges 4–7 were:

Challenge 4 Let the people that advocate Pi calculus show how the Petri net shown in Figure 1 can be modeled easily.

Challenge 5 Let the people advocating Pi calculus propose modeling challenges for people advocating Petri nets as the fundamental language. It would be very interesting the see useful patterns that actually benefit from the notion of mobility present in Pi calculus.

Challenge 6 Let the people that advocate Pi calculus exactly show how existing patterns can be modeled in terms of Pi calculus.

Challenge 7 Let the people advocating Pi calculus propose new patterns,especially patterns involving mobility.

Whatever one's beliefs, it is interesting to try to code the 20 workflow patterns and seeing actual examples will make it easier to make an informed choice of formal foundation.

1.1 Why Even Consider π-calculus?

The π-calculus and its predecessor CCS (Calculus of Communicating Systems) [Mil89] have strong mathematical foundations and have been widely studied for years. This has lead to a plethora of deep theoretical results (see [SW01] for an overview) as well as applications in programming languages (e.g. PICT [PT00]), protocol verification, model checking (e.g. Zing [zin]), hardware design languages[1], and several other areas.

Workflow description languages, it would seem, have a lot in common with these areas. Firstly, workflows can be thought of as parallel processes. Secondly, workflows often defy the block structure found in conventional programming. Thirdly, the prospect of doing formal verification on workflows is very relevant and using a process calculus makes algebraic reasoning and algebraic transformation immediately possible due to the large body of research already present. Lastly, although they are somewhat bare in their style, CCS and π-calculus enforce a very strong separation of process and application logic that seems appropriate for workflow modeling.

On the other hand process calculi are highly theoretical constructs that require a great deal of expert knowledge. Workflow description systems should be accessible to anyone possessing a knowledge of the workflow domain being modeled, and so a significant challenge lies in bridging this gap.

A major strength of the π-calculus is its ability to express passing of channels between nodes and the ability to pass processes (in the higher-order π-calculus, which can be translated down to regular π-calculus). This far, however, only few examples exist that put this capability to use in the workflow domain.

1.2 Contributions

This paper presents a response to challenges 4 and 6, and partial responses to challenges 5 and 7. This leads up to a presentation of SMAWL, the SMAll Workflow Language. More specifically, the contributions are:

1. A CCS encoding of the 20 patterns in Table 1 as suggested in Challenge 6.
2. An brief analysis of the 20 patterns, and a description and encoding of two new patterns: *(8a) N-out-of-M Merge* and *(16a) Deferred Multiple Choice*.
3. SMAWL: a CCS-based compositional language for describing workflows. The language deals with most of the internal message-passing required to code the $20 + 2$ workflow patterns.
4. A short discussion of using CCS vs. π-calculus for workflow description languages.

The challenges suggest the use of π-calculus. It is our opinion that the foundation should be kept as simple as possible, and this research has in fact shown that CCS is sufficient for the purposes of the current workflow patterns.

1.3 Outline

In Section 2 we introduce the particular variant of CCS used throughout the paper. Section 3 offers a brief analysis of the workflow patterns as well as an encoding of the 20 common workflow pattern and the two newly identified ones. Section 4 presents SMAWL and demonstrates it on the *Recording Star* example [wor]. Section 5 contains a discussion and leads for future directions, and Section 6 puts the paper in the context of related work. Section 7 contains a few concluding remarks.

[1] In a sense, a microprocessor is one huge workflow system.

2 Modeling Workflows in CCS

In this paper we will use the CCS syntax defined by the following BNF:

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

0 is the empty process, τ is some unobservable transition, $a!x$ and $a?$ try to send and receive on channel a, $+$ is choice, $|$ runs processes in parallel, $\mathsf{new}\ a\ P$ protects the channel a from communication outside P, and $a?{*}x.P$ spawns the process P each time a message is received on a. The syntax allows unguarded choice but guards all replicated processes with an input prefix ($a?{*}$). For simplicity we often write $a?x$ instead of $a?x.\mathbf{0}$, and we may omit the name x if it is irrelevant in the context. Capital letters, usually P, Q, R, denote processes, and small letters, $a, b, c, \ldots$ denote activities. Internal messages are words in small letters, like *ok* and *go*.

The invocation of an activity a can be encoded as $a!.a?$ denoting a simple request/response mechanism. More sophisticated protocols for monitoring activity progress can be added, but for this paper the statuses started/finished will suffice. Often if a is a workflow activity we will simply write a for $a!.a?$. This will make the workflow activities easier to separate out from the internal message passing.

The non-determinism inherent in CCS should not be considered a problem; rather it is a convenient and elegant method of abstracting from implementation details, application logic or user input. The expression $\tau.P + \tau.Q$ could be abstraction over a data-dependent choice to be made by the system or the decision of a human being.

3 The Workflow Patterns

In this section we consider each of the 20 workflow patterns in turn, discuss their encoding in CCS, and present new patterns. To facilitate comparison the discussion is structured according to the taxonomy given in [vdAtH02] (see Table 1), and the pattern descriptions are taken verbatim from [wor] with minor clarifications in square brackets.

A central part of any workflow formalism is that of splitting the control flow into several possible paths and later joining control flow from different places into fewer paths. This is addressed by the *Basic Control Patterns*, the *Advanced Branching and Synchronization Patterns*, and to a certain extent the *Patterns Involving Multiple Instances*, the *Deferred Choice* pattern, and the *Interleaved Parallel Routing* pattern.

A structured view of these patterns proves beneficial. Consider Table 2 for an overview of Split, Synchronize, and Merge patterns. Split patterns split one path into some m paths. The difference between the split patterns is whether they require control flow to enter only *one* path (choice), *some* paths (multiple choice) or *all* paths (parallel split).

The same trichotomy is useful when considering synchronization and merging. Synchronization waits for a number of signals (one, some or all) and then launches one continuation, whereas merge spawns a continuation for each received signal (at most one, at most some number or for all).

Two new patterns emerge from this exercise: *(16a) Deferred Multiple Choice* and *(8a) N-out-of-M Merge*. These are described and coded below. More generally, one could give every join construct a predicate that dynamically decided when to continue.

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

Table 1. The 20 Workflow Patterns [wor] and Two New Ones

	Split		**Synchronize**		**Merge**	
All	Parallel Split	$1 \to m/m$	Synchronization	$m/m \to 1$	Multiple Merge	$m/m \to m$
One	Exclusive Choice	$1 \to 1/m$	-		Simple Merge	$1/m \to 1$
	Deferred Choice	$1 \to 1/m$	Discriminator	$1/m \to 1$	-	
Some	Multiple Choice	$1 \to n/m$	Synch. Merge	$n/m \to 1$	Multiple Merge	$n/m \to n$
	Deferred Multiple Choice	$1 \to n/m$	N-out-of-M Join	$n/m \to 1$	N-out-of-M Merge	$n/m \to n$

Table 2. Split, Synchronize, and Merge Patterns. $l \to n/m$ means "control flows from l path(s) to n out of m possible paths". $n/m \to l$ means "control flows from n of m possible paths into l.

In the following pattern encodings some internal message-passing is often used for synchronization purposes. Such internal messages, like $done$, ok, $start$, and go, should be concealed to outside expressions through use of the new operator. Except for the first example, we tacitly assume their existence to avoid cluttering the expressions.

Pattern 1. Sequence – *execute activities in sequence.* The first encoding that comes to mind is $a.P$, but unfortunately this dooes not do the job completely. It is desirable to be able to put two arbitrary processes after each other like $P.Q$. A simple solution is provided by Milner [Mil89]: We require all processes to send on an agreed-upon channel, say $done!$, when they are done and the encoding of the sequence $P.Q$ then becomes

$$\mathsf{new}\ go\ (\{^{go}/_{done}\}P \mid go?.Q)$$

where $\{^{go}/_{done}\}P$ is an alpha-conversion that is handled statically on source code level. Henceforth we will use the simpler notation P_{go} to signify that P signals on channel go on completion and that all pre-existing free oks in P have been alpha converted. If the channel is not relevant, it will be omitted. $\square$

Pattern 2. Parallel Split – *execute activities in parallel.*

$$P_1 \mid \cdots \mid P_n \qquad \square$$

Pattern 3. Synchronization – *synchronize two parallel threads of execution.*

$$P_{1,ok} \mid \cdots \mid P_{n,ok} \mid \underbrace{ok?.\cdots.ok?}_{n}.R \qquad \square$$

Pattern 4. Exclusive Choice – *choose one execution path from many alternatives.*

$$\tau.P_1 + \cdots + \tau.P_n$$

The τ transition prefix on each branch allows a data-dependent decision or active decision; that is, the system can decide upon a particular branch (and do away with all other branches) without immediately activating the activities of that branch (cf. *(16) Deferred Choice*). In future patterns involving choice the τ prefix will be omitted unless needed. $\square$

Pattern 5. Simple Merge – *merge two alternative execution paths.*

$$(P_{1,ok} + \cdots + P_{n,ok}) \mid ok?*.R$$

The pattern assumes (unnecessarily for our purpose) that none of the processes P_i are ever run in parallel. This assumption is expressed here by the use of $+$. $\square$

Pattern 6. Multiple Choice – *choose several execution paths from many alternatives.* A simple encoding would be

$$(\tau.P_1 + \mathbf{0}) \mid \cdots \mid (\tau.P_n + \mathbf{0})$$

but this (a) does not enforce a minimum number of activities to be executed and (b) does not explicitly tell if the system is still waiting for a choice to be made or already decided to take the **0** branch. Addressing (b) we get

$$(\tau.P_1 + lazy!) \mid \cdots \mid (\tau.P_n + lazy!)$$

where the system outside should then accept messages on channels ok and $lazy$ appropriately. Addressing (a) amounts to requiring some number of P_is to be started before the system is able to perform rendez-vous on $lazy$. To avoid cluttering up the expression, we do not do this here. $\square$

Pattern 7. Synchronizing Merge – *merge many execution paths. Synchronize if many paths are taken. Simple merge if only one execution path is taken.*

$$(P_{1,ok} + ok!) \mid \cdots \mid (P_{n,ok} + ok!) \mid \underbrace{ok?.\cdots.ok?}_{n}.Q$$

All P_i perform an $ok!$ when they are done so the synchronization mechanism guarding Q should wait for exactly n such messages. □

Pattern 8. Multiple Merge – *merge many execution paths without synchronizing.*

$$(P_{1,ok} + lazy!) \mid \cdots \mid (P_{n,ok} + lazy!) \mid ok?*.Q \mid lazy?*$$

Signals $ok!$ and $lazy!$ signify if an activity was executed or not. Notice that other processes may connect to the merge by using the named entry-point ok. If this in undesired in the context, a new ok can be added around the expression. Additionally, any occurrences of ok in Q should be removed by alpha conversion. □

New Pattern 8a. N-out-of-M Merge – *merge many execution paths without synchronizing, but only execute subsequent activity the first n times. Remaining incoming branches are ignored.*

$$(P_{1,ok} + lazy!) \mid \cdots \mid (P_{n,ok} + lazy!) \mid \underbrace{ok?.go! \mid \cdots \mid ok?.go!}_{n} \mid go?*.Q \mid lazy?*$$

The pattern is similar to *Multiple Merge* except that it is only capable of receiving n messages on ok and then it is done. □

Pattern 9. Discriminator – *merge many execution paths without synchronizing. Execute the subsequent activity only once.*

$$(P_{1,ok} + lazy!) \mid \cdots \mid (P_{n,ok} + lazy!) \mid ok?.(Q \mid lazy?* \mid ok?*)$$

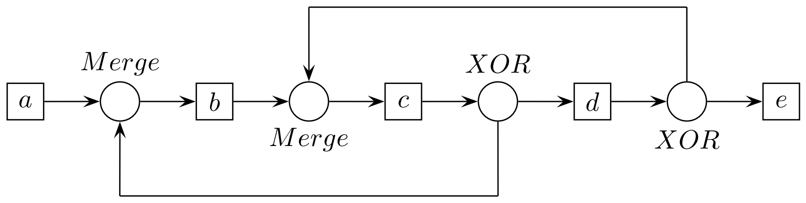
Here the *Multiple Choice* pattern is used but other split patterns could be used too. The first signal on ok initiates Q and all remaining signal are consumed. Contrary to Petri nets, ignoring all future signals is not a problem because any loop around this construct would simply instantiate a new process (with new channel names) in each iteration. □

Pattern 9a. N-out-of-M Join – *merge many execution paths. Perform partial synchronization and execute subsequent activity only once.*

$$(P_{1,ok} + lazy!) \mid \cdots \mid (P_{n,ok} + lazy!) \mid \underbrace{ok?.\ldots.ok?}_{n}.(Q \mid lazy?* \mid ok?*)$$

For the purpose of this demonstration, the pattern is combined with *Multiple Choice* pattern. Once n messages are received on ok, the process Q is activated and all remaining messages, whether ok or $lazy$, are discarded. □

Pattern 10. Arbitrary Cycles – *execute workflow graph w/out any structural restriction on loops.* Consider the following example from [wor]:



Since activities b and c have multiple merge points that do not originate from the same split block, it is necessary to promote them to named services (or "functions") by using the replication operator – a lot like what one would do in any structured programming language. Here we name them go_b and go_c and the example becomes

$$a.go_b! \mid go_b?*.b.go_c! \mid (go_c?*.c.(go_b! + d.(go_c! + e))). \qquad \square$$

Pattern 11. Implicit Termination – *terminate if there is nothing to be done.*

Implicit termination means detecting if all processes have come to a stable state and there are no pending activities or messages. In other words if the residual workflow process – disregarding subexpressions of the form $a?*.P$ – is structurally congruent to **0**. Subexpression of the form $a?*.P$ can be thought of as functions; they cannot and should not be reduced. Any active invocations of such a function will manifest themselves as some reduced form of the body P. (However, if a function listens on a channel that cannot be reached by the active process expression, it may be garbage collected by the runtime system, but this is separate from detecting implicit termination.)

Explicit termination was modeled in the *Sequence* pattern where we adopted the convention that all processes signal on a pre-determined channel on completion. When the top-level expression wishes to send its signal, the process explicitly terminates. □

Pattern 12. MI without synchronization – *generate many instances of one activity without synchronizing them afterwards.*

$$P.loop! \mid loop?*.(create!.loop! + R) \mid create?*.Q$$

After process P finishes an arbitrary number of instances of Q are spawned and once no more Qs need to be started, R is executed. Execution of R does not wait for the completion of any of the instances of Q. The choice inside the loop can be an *Exclusive Choice* or a *Deferred Choice*. □

Pattern 13. MI with a priori known design time knowledge – *generate many instances of one activity when the number of instances is known at the design time (with synchronization).*

$$\underbrace{create!.\cdots.create!}_{n}.\underbrace{ok?.\cdots.ok?}_{n}.Q \mid create?*.P_{ok}$$

This pattern creates exactly n instances of P and waits for all of them to complete before passing control on to the process Q. □

Pattern 14. MI with a priori known runtime knowledge – *generate many instances of one activity when a number of instances can be determined at some point during the runtime (as in FOR loop but in parallel).* First a counter is needed to keep track of the number of instances that need to be synchronized. It can be coded (a) directly in CCS through cunning use of the `new` operator (cf. sect. 7.5 of [Mil89]), (b) by adding a notion of simple datatypes such as integers to CCS or (c) by using channel-passing π-calculus style. For now we vil just add a process $Counter$ and not bother with its implementation details; the important part being that is it feasible in CCS. $Counter$ has the ability to receive messages on channel inc (increase) and to send message on channels dec (decrease) or $zero$ (check if zero) depending on its state:

$$\begin{aligned} P.loop! \;\mid\; & loop?*.(create!.loop! + break!) \mid create?*.inc!.Q_{ok} \\ \mid\; & break?*.(ok?.dec?.break! + zero?.R) \mid Counter \end{aligned}$$

From P the process enters a loop where multiple instances of Q are created. Once creation is done, the process iterates the $break$ loop until a message on $zero$ is received, i.e. 0 instances remain. Notice that one can decide not to create any instances at all., if one desires a minimum number of processes to be spawned, the $break!$ prefix should be guarded by unfolding the $create$ loop an appropriate number of times. □

Pattern 15. MI with no a priori runtime knowledge – *generate many instances of one activity when a number of instances cannot be determined (as in WHILE loop but in parallel).* This pattern is merely a simplified version of pattern 14. There is no longer a need to distinguish between the creation phase and the synchronization phase. Hence it collapses to:

$$\begin{aligned} P.loop! \;\mid\; & loop?*.(create!.loop! + ok?.dec?.loop! + zero?.R) \mid create?*.inc!.Q_{ok} \\ \mid\; & Counter \end{aligned}$$

More advanced synchronization schemes can be plugged in at this place. Maybe only som n instances need to finish, maybe the completion condition is some predicate ρ over the data produced so far. In other words, the logic deciding this loop can be pushed up to a data-aware layer or handled through more complex join conditions in the process expression. □

Pattern 16. Deferred Choice – *execute one of the two alternatives threads. The choice which thread is to be executed should be implicit.*

$$P_1 + \cdots + P_n \quad \text{where each subprocess is guarded.}$$

This is very similar to the *Exclusive Choice*. Here the choice is made exactly when a (non-silent) transition of either of the P_is occurs, and hence we require all P_i to be guarded. In the *Exclusive Choice* we might have $\tau.P + \tau.Q$ to signify that an abstract upper-layer would decide which branch to follow (i.e. what τ transition to take). □

New Pattern 16a. Deferred Multiple Choice – *execute several of the many alternative threads. The choice of which threads are to be executed should be implicit.* The important question here is: how do we know when the users are done choosing the threads?

Two approaches are possible: (a) we fix an n number of threads to be activated or (b) we set up a virtual activity that means “all desired threads have been started, remove remaining choices”. Option (a) requires all subprocesses to emit a signal on activation and consume n such signals before activating a $lazy?{*}$process to remove the remaining branches. Here we follow option (b):

$$(P_1 + lazy!) \mid \cdots \mid (P_n + lazy!) \mid donechoosing.lazy?{*}$$

Firstly, this pattern cannot be obtained by combination of several *Deferred Choice Patterns*. For instance, the expression $(A.(B{+}donechoosing){+}B.(A{+}donechoosing){+}donechoosing)$, does not allow the second activity to be started before the first one is done. □

Pattern 17. Interleaved Parallel Routing – *execute two activities in random order, but not in parallel.*

$$\begin{aligned}&\mathsf{new}\ lock, unlock, locked, unlocked\ (lock!.P_1.unlock! \mid \cdots \mid lock!.P_n.unlock!\\ &\mid (unlocked? * .lock?.locked! \mid locked? * .unlock?.unlocked! \mid unlocked!))\end{aligned}$$

This is generalized slightly from two to any finite number of interleavings. Each process P_i acquires the lock on activation and releases it upon termination. □

Pattern 18. Milestone – *enable an activity until a milestone is reached.* Assume that process Q can only be enabled after P has finished and only before R is started. This is done by the help of a small flag that can be changed using $set!$ and $clear!$ and tested using $ison?$ and $isoff?$:

$$\begin{aligned}Milestone(ison, isoff, set, clear) = {}& on?{*}.(ison!.off! + set?.on! + clear?.off!) \mid\\ & off?{*}.(isoff!.off! + set?.on! + clear?.off!) \mid off!\end{aligned}$$

Now the milestone can be set up as

$$P.set!.{}_{clear}R \mid ison?{*}.Q \mid Milestone$$

where ${}_{clear}R$ denotes the modification of process R that signals on $clear$ when it starts its first activity or makes its first explicit choice. □

Pattern 19. Cancel Activity – *cancel (disable) an enabled activity.* Suppose there are activities a, b, and c being carried out in sequential order. Anytime during the execution of activity b the process can be cancelled. We break up activity b into $b!.b?$ so it will be clear exactly when a cancellation can be accepted. Assume cancellation is obtained by signaling on channel $cancel$:

$$a.b!.(b?.c + cancel?)$$ □

Pattern 20. Cancel Case – *cancel (disable) the process.* The example of this pattern given at [wor] is an online travel reservation system. If the reservation for a plane ticket fails, then all pending reservations for flights, hotel, car rental, etc. in the itinerary must be cancelled immediately to avoid unnecessary work.

Intuitively, this corresponds to returning to some predefined state in the system after relinquishing all resources and information bound in the current context. In this respect the cancellation patterns look at lot like a program exceptions, and indeed the cancellation pattern can be coded in CCS by using a source code transformation.

Pattern 19 showed how to insert a cancellation point a one particular spot in a process. We now insert cancellation points for all states in the process $a.b.c$ to obtain

$$a!.(a?.(b!.(b?.(c!.(c?+cancel?)+cancel?)+cancel?)+cancel?)+cancel?)+cancel?$$

Clearly this becomes very tedious and strongly suggests that a more abstract language would be useful. □

4 Towards a Workflow Description Language Based on CCS

The workflow patterns coded in the previous sections show that CCS is a bit too low-level to be used directly as a workflow language, and that certain patterns are very cumbersome to write. We therefore add a number of combinators and, in the process, give the language a more pleasing syntax. The goals are (a) to reduce the amount of user-specified internal synchronization mechanisms and (b) to provide elegant constructs for the 22 workflow patterns.

A natural idea is to make small building blocks of each of the patterns we have seen to far. This way there would be, e.g., a number of split combinators and a number of join combinators. However, it turns out to 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:

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

In the syntax ϵ denotes the empty string, f is a function name, w is the workflow name, ρ is a data-dependent predicate [2], 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.

[2] 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.

A program consists of a pair (W, D), i.e. a named workflow and a list of function/milestone/newlock declarations. Arbitrary loops are encoded through named functions. Milestones are defined separately and can be read/set by any process knowing the correct channels. Parallel interleaving is handled through global locks. This means that two processes that are not allowed to run at the same time, do not have to be within the same logical block.

The construct **choose any (wait for** k**)**$\{PP$ **merge**(n) $P\}$ implements multiple choice over the PPs, then merges each of the threads to P, 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 P upon completion. A value of $n \neq \infty$ signifies that only the first n threads to complete should give rise to an instantiation of P. If **merge**$(n)P$ is missing, n is taken to be 0 and P can be anything. The construct **do all** implements parallel split with the same options of combining with merge and synchronize patterns. The remaining constructs should be self-explanatory.

It may be helpful to consider an example. The example shown in Figure 1 is based loosely on the Petri net example at [wor] to make it easier to compare the Petri net approach to a CCS approach.

4.1 Compiling SMAWL to CCS

Compiling the workflow description language is a relatively easy task since the language is based directly on patterns that have already been described in CCS. The main transformation $\mathcal{T}[\![\cdot]\!]$ is a map $W \cup D \rightarrow Channel \rightarrow CCS$, where $Channel$ denotes the set of valid channel names. The following auxiliary function are needed: mergeprefix is the map $\mathbb{N} \cup \{\infty\} \times Channel \times Channel \rightarrow CCS$ defined by:

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

The function $\nu : \{()\} \rightarrow Channel$ returns a fresh channel name that has not previously been used on every call.

The transformation of the **cancel** construct makes use of the function $\mathcal{C}[\![\cdot]\!]$, which insert cancellation point at all possible states as discussed in the **Cancel Case** pattern. Formally, the cancellation transformation $\mathcal{C}[\![\cdot]\!]$ on CCS expressions is expressed as:

$$\begin{aligned} \mathcal{C}[\![\mathbf{0}]\!] &= \mathbf{0} \\ \mathcal{C}[\![\alpha.P]\!] &= \alpha.\mathcal{C}[\![P]\!] + cancel? \\ \mathcal{C}[\![P + Q]\!] &= \mathcal{C}[\![P]\!] + \mathcal{C}[\![Q]\!] + cancel? \\ \mathcal{C}[\![P \mid Q]\!] &= \mathcal{C}[\![P]\!] \mid \mathcal{C}[\![Q]\!] + cancel? \\ \mathcal{C}[\![\mathsf{new}\ \ a\ P]\!] &= \mathsf{new}\ \ a\ \mathcal{C}[\![P]\!] + cancel? \\ \mathcal{C}[\![a?{*}x.P]\!] &= (a?{*}x.\mathcal{C}[\![P]\!]) + cancel? \end{aligned}$$

```
workflow Become a recording star =
    choose one {
        ⇒ call (Work your way up)
        ⇒ call (Try to get lucky)
    };
    Make record;
    do all {
        ⇒ choose one {
            ⇒ Develop as an artist
            ⇒ Develop bad habits
            }
        ⇒ Rehearse tour;
            Do tour
    }
end
```

```
fun Work your way up =
    multi {Learn to play};
    choose one {
        ⇒ Join a band
        ⇒ Decide to go solo
    };
    choose any {
        ⇒ multi {Write song}
        ⇒ multi {Perform live}
    }
end

fun Try to get lucky =
    Do audition;
    choose one {
        ⇒ $Audition.failed$
            choose one {
                ⇒ call (Try to get lucky)
                ⇒ call (Work your way up)
            }
        ⇒ $Audition.passed$
            Do everything you are told
    }
end
```

Fig. 1. A Larger Example: How To Become a Recording Star

where $alpha$ denotes any prefix τ, $a?x$, or $a!x$.

Given a program (W, D), the CCS expression representing the program is obtained as $\mathcal{T}[\![W]\!]abort \mid \mathcal{T}[\![D]\!]dontcare$. Incidentally, the transformation also shows, that new operators can statically be removed bar the cases where new is used inside replicated processes such as the $Counter$ processes.

4.2 Challenge 4 and the Overlaying Combinator

What remains is a tiny – somewhat speculative – improvement to the language. Consider the Petri net in Figure 3. The Petri net exhibits a pattern that is cumbersome in relation to our goal of get rid of user-defined internal message-passing, namely it defies the block structure upon which the combinators are based so far.

The Petri net can be mimicked in CCS through use of function definitions and explicit message-passing, but more interestingly it can be expressed elegantly using the overlay operator $\&$ defined by the rules in Figure 4. It is a somewhat inspired by the $||$ operator found in CSP [Hoa85]. The operator is perhaps best explained by coding the example in Figure 3:

$$C4 = a.(b.c.d|e.f.g).h \;\&\; (c|e).f$$

$$
\begin{aligned}
&\mathcal{T}\llbracket \textbf{workflow}\ w = P\ \textbf{end} \rrbracket = \mathcal{T}\llbracket P \rrbracket\\
&\mathcal{T}\llbracket \textbf{fun}\ f = P\ \textbf{end}\ D \rrbracket = \lambda ok.f?{*}.\mathcal{T}\llbracket P \rrbracket f \mid \mathcal{T}\llbracket D \rrbracket ok\\
&\mathcal{T}\llbracket \textbf{milestone}(ison, isoff, set, clear)\ D \rrbracket =\\
&\quad \lambda ok.\text{Milestone}(ison, isoff, set, clear) \mid \mathcal{T}\llbracket D \rrbracket ok\\
&\mathcal{T}\llbracket \textbf{newlock}(l,u) \rrbracket = \lambda ok.\text{let } unlocked \Leftarrow \nu() \text{ in let } locked \Leftarrow \nu() \text{ in}\\
&\quad unlocked? * .l?.locked! \mid locked? * .u?.unlocked! \mid unlocked!\\
&\mathcal{T}\llbracket \textbf{lock}(l,u)\{P\} \rrbracket = \lambda ok.\text{let } ok' \Leftarrow \nu() \text{ in } l!.\mathcal{T}\llbracket P \rrbracket ok' \mid ok'?.u!.ok!\\
&\mathcal{T}\llbracket activity \rrbracket = \lambda ok.activity!.activity?.ok!\\
&\mathcal{T}\llbracket \textbf{send}\ f \rrbracket = \lambda ok.f!.ok!\\
&\mathcal{T}\llbracket \textbf{receive}\ f \rrbracket = \lambda ok.f?.ok!\\
&\mathcal{T}\llbracket P;Q \rrbracket = \lambda ok.\text{let } ok' \Leftarrow \nu() \text{ in } \mathcal{T}\llbracket P \rrbracket ok' \mid ok'?.\mathcal{T}\llbracket Q \rrbracket ok\\
&\mathcal{T}\llbracket \rho\ P \rrbracket = \lambda ok.\tau.\mathcal{T}\llbracket P \rrbracket ok\\
&\mathcal{T}\llbracket \textbf{choose one}\ \{\Rightarrow P_1 \Rightarrow \cdots \Rightarrow P_n\} \rrbracket = \lambda ok.\mathcal{T}\llbracket P_1 \rrbracket ok + \cdots + \mathcal{T}\llbracket P_n \rrbracket ok\\
&\mathcal{T}\llbracket \textbf{choose any (wait for } \text{k}\textbf{)}\ \{\Rightarrow P_1 \Rightarrow \cdots \Rightarrow P_n\ (\textbf{merge}\ l\ Q)\} \rrbracket =\\
&\quad \lambda ok.\text{let } ok' \Leftarrow \nu() \text{ in let } ok'' \Leftarrow \nu() \text{ in let } ok''' \Leftarrow \nu() \text{ in}\\
&\qquad (\mathcal{T}\llbracket P_1 \rrbracket ok' + lazy!) \mid \cdots \mid (\mathcal{T}\llbracket P_n \rrbracket ok' + lazy!) \mid lazy?{*}\\
&\qquad \mid \text{mergeprefix}(l, ok', ok'') \mid ok''?{*}.\mathcal{T}\llbracket Q \rrbracket ok''' \mid \underbrace{ok'''?.\cdots.ok'''?}_{k}.(ok! \mid ok'''?{*})\\
&\mathcal{T}\llbracket \textbf{do all (wait for } \text{k}\textbf{)}\ \{\Rightarrow P_1 \Rightarrow \cdots \Rightarrow P_n\ (\textbf{merge}\ l\ Q)\} \rrbracket =\\
&\quad \lambda ok.\text{let } ok' \Leftarrow \nu() \text{ in let } ok'' \Leftarrow \nu() \text{ in let } ok''' \Leftarrow \nu() \text{ in}\\
&\qquad \mathcal{T}\llbracket P_1 \rrbracket ok' \mid \cdots \mid \mathcal{T}\llbracket P_n \rrbracket ok'\\
&\qquad \mid \text{mergeprefix}(l, ok', ok'') \mid ok''?{*}.\mathcal{T}\llbracket Q \rrbracket ok''' \mid \underbrace{ok'''?.\cdots.ok'''?}_{k}.(ok! \mid ok'''?{*})\\
&\mathcal{T}\llbracket \textbf{call}\ f \rrbracket = \lambda ok.f!.f?.ok!\\
&\mathcal{T}\llbracket \textbf{cancel}\ \{P\} \rrbracket = \lambda ok.\mathcal{C}\llbracket \mathcal{T}\llbracket P \rrbracket ok \rrbracket\\
&\mathcal{T}\llbracket \textbf{multi}(n)\ \{P\} \rrbracket = \lambda ok.\text{let } create \Leftarrow \nu() \text{ in let } ok' \Leftarrow \nu() \text{ in}\\
&\quad \underbrace{create!.\cdots.create!}_{n}.\underbrace{ok'?.\cdots.ok'?}_{n}.ok! \mid create?{*}.\mathcal{T}\llbracket P \rrbracket ok'\\
&\mathcal{T}\llbracket \textbf{multi}(\infty)\ \{P\} \rrbracket = \lambda ok.\text{let } inc \Leftarrow \nu() \text{ in let } dec \Leftarrow \nu() \text{ in let } zero \Leftarrow \nu() \text{ in}\\
&\quad \text{let } loop \Leftarrow \nu() \text{ in let } ok' \Leftarrow \nu() \text{ in let } create \Leftarrow \nu() \text{ in}\\
&\quad loop! \mid loop?{*}.(create!.loop! + zero?.ok! + ok'?.dec?.loop!)\\
&\quad \mid create?{*}.inc!.\mathcal{T}\llbracket P \rrbracket ok' \mid Counter(inc, dec, zero)
\end{aligned}
$$

Fig. 2. The Transformation $\mathcal{T}\llbracket\cdot\rrbracket$ from SMAWL to CCS

Whenever $C4$ wants to transition on a channel that is mentioned in both operands to the $\&$ operator, both operands have to agree to this. If a transition is only mentioned in one of the operands, the $\&$ operator has no effect on that transition. Here if a, e was

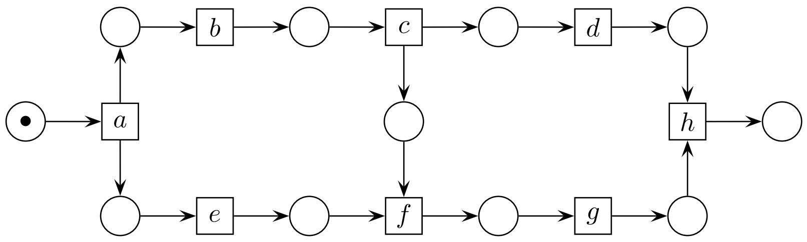


Fig. 3. Challenge 4 [vdA04]

performed first, the left process would be ready to do f but the right would not, since it would need c to complete first.

The operator is very interesting because it allows the overlaying of global business rules to all processes. E.g. one might have a rule that says “delivery shall occur invoicing”. This rule would now simply be enforced on all workflows by overlaying it with the workflows. Also, the workflow designers do not have to describe all business rules because the can simply overlay the global rules to their workflows.

The operator can be built into the reduction rules of a workflow system or it can be translated down to regular CCS, which, unfortunately, can generate an exponential blowup in expression size.

$$\frac{P \xrightarrow{\alpha} P' \quad \alpha \notin \text{ports}(Q)}{P\&Q \xrightarrow{\alpha} P'\&Q}\text{(+sym.)} \qquad \frac{P \xrightarrow{\alpha} P' \quad Q \xrightarrow{\alpha} Q'}{P\&Q \xrightarrow{\alpha} P'\&Q'}$$

Fig. 4. Reduction Rules for the Overlaying Operator &

5 Discussion and Future Work

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 tool support for SMAWL and equally it would be interesting to plug SMAWL into a formal verification tool.

The two new patterns *(16a) Deferred Multiple Choice* and *(8a) N-out-of-M Merge* should be expressed in Petri nets, and likewise it would be interesting to see if Petri nets can handle the overlaying operator more elegantly than CCS. An investigation into the pratical applicability of the overlaying operator $\&$ might prove very fruitful.

When debating expressiveness an easy, but vacuous point is Turing completeness. Indeed, both Petri nets and CCS are Turing complete, but in their raw form they are unsuited for any non-trivial programming/modeling task. For domain-specific languages it would seem more reasonable to consider criteria such as (a) ontological fit (ie. domain closeness, convenience of expressing common idioms), (b) expressiveness, (c) amenability to formal analysis, (c) simplicity, and, for workflow languages, (d) graphical representation.

To explore expressiveness properly, it would be interesting to put all the languages into a formal semantic framework. For workflows, the idea of representing workflows as sets of allowable traces CSP-style seems reasonable and will provide a common ground for comparison. First the patterns should be described in a more formal fashion and then the expressiveness of Petri nets, CCS, CSP, temporal logic and others can be compared using e.g. Felleisen beautiful characterization of expressiveness [Fel90][3].

π-calculus mobility was not needed. Essentially, workflow systems deal with the flow of command/control and not as much the channels for exchanging messages, if one desires to shift focus to the flow of data—and in particular which channels are used for this—then maybe π-calculus is appropriate.

6 Related Work

Dong and Shensheng presented their encoding of some of the patterns in an unpublished paper in 2004 [DS]. Their encodings differ a great deal from the ones presented here, but more fundamentally they use the channel-passing facility of π-calculus, whereas the encodings here do not and thus can be expressed in CCS. In our view the language should be as simple as possible, and as we have demonstrated, the full power of π-calculus is not needed. Also, staying within CCS makes the patterns more amenable to use in the tools current available.

Numerous attempts to map Petri nets to process calculi exist. Also an effort to merge the best of both worlds exists (cf. Petri Box Calculus [BDK01]).

7 Conclusion

The purpose of this paper was to show that workflow patterns in fact can be expressed in CCS. We demonstrated how to code challenges 4 and 6, identified new patterns, and presented a language, that makes CCS-based workflow systems more accessible.

The response to challenges 5 and 7 is necessarily this: The coding of the 20 patterns has provided no compelling reasons to use the channel-passing mechanism of π-calculus. We see no need for the notion of mobility found in π-calculus for workflow systems. Workflow systems abstract away from the actual channels of delivery. Maybe π-calculus is relevant when analyzing the interaction between agents and clients inside the workflow activities.

In remains open if the newly discovered patterns occur often enough in pratical settings to warrant their incorporation with the rest. Real world examples to support them would be highly appreciated – especially for the overlaying operator.

SMAWL turned out to be an easy and very convenient language for writing workflow expressions and more work will conducted in this direction.

[3] Quoting Felleisen [Fel90]: ... “more expressive” means that the translation of a program with occurrences of one of the constructs c_i to the smaller language requires a global reorganization of the entire program.

References

[BDK01] Eike Best, Raymond Devillers, and Maciej Koutny. *Petri Net Algebra*. Monographs on theoretical computer science. Springer, 2001.

[DS] Yang Dong and Zhang Shensheng. Modeling workflow patterns with pi-calculus. Unpublished. Available from http://www.workflow-research.de.

[Fel90] Matthias Felleisen. On the expressive power of programming languages. *Science of Computer Programming*, 1990.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. International Series in Computer Science. Prentice-Hall, 1985.

[Mil89] Robin Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice-Hall, 1989.

[PT00] 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.

[SW01] D. Sangiorgi and D. Walker. *The π-calculus: a Theory of Mobile Processes*. Cambridge University Press, 2001.

[vdA04] W.M.P. van der Aalst. Pi calculus versus petri nets: Let us eat ”humble pie” rather than further inflate the ”pi hype”. 2004.

[vdAtH02] 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.

[wor] Workflow patterns. Available from http://www.workflowpatterns.com.

[zin] The zing model checker. Available from http://research.microsoft.com/zing/.

[zM] Michael zur Muehlen. Workflow research. http://www.workflow-research.de.