Abstract

This talk sketches some recent research toward a declarative process-based framework for ERP systems. In particular, the talk describes the following two areas:

(1) A declarative language for compositional specification of contracts governing the exchange of resources. The language extends Eber and Peyton Jones’s declarative language for specifying financial contracts to the exchange of money, goods and services amongst multiple parties, and it complements McCarthy’s Resources, Events and Agents (REA) accounting model with a view-independent formal contract model that supports definition of user-defined contracts, automatic monitoring under execution, and user-definable analysis of their state before, during and after execution.

(2) A pi-calculus encoding of common workflow control patterns, which leads to a pi-calculus-based macro language for workflow specification. The encodings presented here demonstrate some of the strengths and weaknesses of pi-calculus for business process formalization vis a vis Petri nets and concrete languages such as BPEL and YAWL.
The NEXT Research Group

- NEXT generation Enterprise Resource Planning (ERP) systems
- "The over-all goal of the project is to develop software technology and methods for development of the next generation of business management systems for companies. These systems are often referred to as ERP-systems (ERP: Enterprise Resource Planning) even though today they comprise much more than resource planning."

- Fritz Henglein
- Henrik Reif Andersen
- Peter Sestoft
- Rune Møller Jensen
- Kasper Østerbye
- Peter Carstensen
- Yvonne Dittrich

What Is an Enterprise System?

- Usually centered around a general ledger, enterprise systems handle business by supporting:
  - Accounts payable/receivable
  - Inventory, assets, projects, resource allocation
  - Employees, workflows, tasks
  - Customer relations
  - Supply-chain management, logistics
The Goal

- To be able to write, automate, analyze and monitor business processes in a service-oriented architecture.

- Current systems (SAP, PeopleSoft, Axapta, Siebel, Great Plains, Compiere) are not process-oriented:
  - ad hoc-integration between modules,
  - do not expose and share processes with the environment,
  - do not support mobility.

The Project

- Design a process-oriented programming platform.

- Processes, constraints, and rules
- Data model
- Reporting and monitoring
Parts of a Contract

- Definitions
  (define parties, nature and quality of exchanged resources, legal context, exception handlers etc.)

- Temporal/logic structure

- **The good news:** We can express temporal properties formally.
- **The bad news:** Probably cannot eliminate the need for lawyers altogether!
Paper-Based Contract-Handling is Costly

- No formal representation results in:
  - Manual handling in auxiliary systems
  - Ad hoc deadline management
  - Ambiguous semantics
  - Time-consuming valuation
  - Cannot report on future events
  - No coordination with production planning or supply-chain management
  - Missed opportunities (call options etc.)

- Potential benefits of formal representation:
  - Alleviate the problems above
  - Several consistency checks at time of writing contract
  - (Semi-)automated contract analysis (incl. valuation)
    (DSL programs are in a sense “intelligent data”)

We Analyzed 15 Full-Length Contracts

- Data model
- Structure
Example: Agreement to Provide Legal Services

- **Section 1.** The attorney shall provide, on a non-exclusive basis, legal services up to (n) hours per month, and furthermore provide services in excess of (n) hours upon agreement.

- **Section 2.** In consideration hereof, the company shall pay a monthly fee of (amount in dollars) before the 8th day of the following month and (rate) per hour for any services in excess of (n) hours 40 days after the receipt of an invoice.

- **Section 3.** This contract is valid 1/1-12/31, 2008.

Primitives in Contracts

- **Data**
  - Agents
  - Resources (goods, services, rights)
  - Commitments/Events
  - Time

- **Structure**
  - Sequential execution
  - Concurrent execution
  - Repeated execution
  - Alternative execution (choice between subcontracts)
Syntax

Success/Failure

The succeeded/failed contract with no commitments.

\( \text{transmit}(A_1, A_2, R, T, P, c) \)

- The commitment of agent \( A_1 \) to transmit resource \( R \) to agent \( A_2 \) at time \( T \) subject to predicate \( P \) (afterwards do contract \( c \)).

\( c_1 ; c_2 \)

- A sequence of two contracts. The first contract must be reduced to Success before the second can begin.

\( c_1 \parallel c_2 \)

- Parallel, independent execution of two contracts.

\( c_1 + c_2 \)

- (Non-deterministic) choice between two contracts.

\( f(\bar{a}) \)

- Expansion to body of contract \( f \) with arguments \( \bar{a} \).

\( \text{letrec}\ f_i[X_i] = e_i\ \text{in}\ c \)

- Contract \( c \) with named contracts \( f_i \) with formal arguments \( X_i \) bound to \( e_i \).

Example: Legal Agreement Code

```plaintext
letrec
extra (att, com, invoice, pay) =
  (Success
   + transmit (att, com, invoice, T2).
   transmit (com, att, pay, T3 | T3 <= T2 + 45d))

legal (att, com, fee, invoice, pay, n, m, end) =
  transmit (att, com, H, T | n < T and T <= m).
  (extra (att, com, invoice, pay)
  || transmit (com att, fee, T | T <= m + 8d)
  || (legal (att, com, fee, invoice, pay, m, min(m + 30d, end)
   + transmit (att, com, end, T | end <= T)))
  in
  legal ("Attorney","Company",10000,invoice,pay,0,30,360)
```

When Is a Contract Satisfied?

- Contracts denote sets of finite traces. A trace is a finite sequence of events:
  \[ s ::= <> \mid \text{transmit}(a1,a2,r,t) \ s \]

- So contracts classify a given trace as performing or nonperforming. Formally:
  \[ C[c]^{D[D]^\delta \oplus \delta'} = \{ s \mid \delta' \vdash_D^\delta s : c \} \]

The Satisfaction Relation

- How about Failure? (Exercise for the audience)
- Note that the base language is orthogonal to this. Just plug in your own.
From the Satisfaction Relation to Denotational Semantics

- Denotational semantics, e.g.:
  \[ C[c_1 + c_2]^{\gamma:\delta} = C[c_1]^{\gamma:\delta} \cup C[c_2]^{\gamma:\delta} \]

- Theorem: Denotational characterization of satisfaction
  \[ C[c]^{\mathcal{P}[D]^{\delta:\delta'}} = \{ s' \mid \delta' \models_D s : c \} \]

Toward Contract Execution

- We need a representation of residual contracts in order to monitor execution.

- For a trace set \( S \) and an event \( e \), define the residuation function as:
  \[ e \setminus S = \{ s' \mid \exists s \in S : es' = s \} \]
Residual Contracts Should Be Expressible

- Let $Sc$ denote the trace set of contract $c$.
- Not immediately clear if $e \setminus Sc$ is denotable by a contract (syntactical expression) $c'$
- But this is clearly a normative property!

Not All Residuals Are Representable

\begin{align*}
\text{letree } f(N) &= (\text{transmit}(a_1, a_2, r, T \mid T \leq N) \parallel f(N + 1)) \text{ in } f(0) \\
\text{Assume event transmitting } (a_1, a_2, r, 0) \text{ occurs.} \\
\text{transmit}(a_1, a_2, r, T_0[T_0 \leq 0) \parallel \cdots \parallel \text{transmit}(a_1, a_2, r, T_i[T_i \leq i) \parallel \cdots}
\end{align*}

We therefore introduce guardedness, a sufficient condition to ensure that all contracts have a syntactic representation of the residual contract under any event.
Guarded Contracts

- Intuitively, a contract is guarded if (mutual) recursive calls are prefixed by a transmit.

\[
\begin{align*}
\text{D} \vdash \text{Success guarded} & \quad \text{D} \vdash \text{Failure guarded} \\
\text{D} \vdash \text{transmit}(X \mid P).c \text{ guarded} & \quad \text{D} \vdash c \text{ guarded} \quad (f(X) = c) \in D \\
\text{D} \vdash c \text{ guarded} & \quad \text{D} \vdash c' \text{ guarded} \\
\text{D} \vdash c + c' \text{ guarded} & \quad \text{D} \vdash c \parallel c' \text{ guarded} \\
\text{D} \vdash c \text{ nullable} & \quad \text{D} \vdash c \text{ guarded} \\
\text{D} \vdash c \text{ guarded} & \quad \text{D} \vdash c' \text{ guarded} \\
\text{D} \vdash c; c' \text{ guarded} & \quad \text{D} \vdash c; c' \text{ guarded} \\
\text{D} \nvdash c \text{ nullable} & \quad \text{D} \vdash c \text{ guarded} \\
\text{D} \vdash c \text{ guarded} & \quad \text{D} \vdash c; c' \text{ guarded}
\end{align*}
\]

Several Operational Semantics

- Now that guarded contracts are representable, we can attempt defining an operational semantics to govern contract rewriting. We defined three inter-related semantics:

  - Non-deterministic eager matching
  - Deterministic eager matching with explicit routing
  - Deterministic reduction by delayed matching
Deterministic Reduction with Delayed Matching

• Consider some of the reduction rules:

\[ \delta \oplus \{ X \mapsto v \} \models P \quad (v = Q[a]^{\delta}) \]

\[ D, \delta \vdash_D \text{transmit}(X \mid P), c \overset{\text{transmit}(v)}{\longrightarrow} c[v/X] \]

\[ D, \delta \vdash D \quad c \overset{e}{\rightarrow} d\]

\[ D, \delta \vdash_D c' \overset{e}{\rightarrow} d' \]

\[ D, \delta \vdash_D c + c' \overset{e}{\rightarrow} d + d' \]

\[ D, \delta \vdash_D c \parallel c' \overset{e}{\rightarrow} d\parallel d \parallel d' \]

Guardedness Ensures Safe Residuation

• Guarded Subject Reduction:

For any \( c, c', \delta, e \) and \( D \): if \( D, \delta \vdash_D c \overset{e}{\rightarrow} c' \) then \( D, \delta \models e \vdash c = c' \).

For all \( c, \delta \) and guarded \( D \), there exists a unique \( c' \) such that \( D, \delta \vdash_D c \overset{e}{\rightarrow} c' \); \( D \vdash e' \) guarded.

• Now we reap the benefits of compositionality: all residual contracts have syntactical representations and can be submitted to analysis etc.
Example Reduction

transmit (att, com, H, T | 0 < T and T <= 30).
( transmit (com, att, fee, T | T <= 30 + 8d)
 || ( legal (... , 30, min(30 + 30d,60), 60)  
    + transmit (att, com, end, T | 60 <= T)))

Services rendered first month:

\[(att, com, h1, 20)\]

\[\]

( transmit (com, att, fee, T | T <= 30 + 8d)
 || ( legal (... , 30, min(30 + 30d,60), 60)  
    + transmit (att, com, end, T | 60 <= T)))

Services rendered second month:

\[(att, com, h2, 37)\]

Example Reduction - Step 2

\[
( \text{transmit (com, att, fee, T | T \leq 30 + 8d)}
\]
\[
\| ( \text{legal (..., 30, min(30 + 30d,60), 60)}
\]
\[
+ \text{transmit (att, com, end, T | 60 \leq T)})
\]

Services rendered second month:

\[(att, com, h2, 37)\]

\[
(\text{Failure}
\]
\[
\| ( \text{legal (..., 30, min(30 + 30d,60), 60)}
\]
\[
+ \text{transmit (att, com, end, T | 60 \leq T)})
\]

+ ( transmit (com, att, fee, T | T <= 30 + 8d)
\]
\[
\| ( \text{transmit (com, att, fee, T | T \leq 60 + 8d)}
\]
\[
\| ( \text{legal (..., 60, min(60 + 30d,60), 60)}
\]
\[
+ \text{transmit (att, com, end, T | 60 \leq T)})
\]

+ Failure)
Contract Analysis

- Compositional and thus available at runtime too!
- Wish to answer questions ranging from simple (what is my current todo list?) to complex (what is the current value of the contract based on my stochastic model?)
- Particularly important analysis: Failure, i.e. does the contract have no satisfying traces.

\[ \forall \delta', \forall t' \geq t : (\delta \oplus \delta' \oplus T \rightarrow t' \models \neg P) \]
\[ \text{D, } \delta, t \vdash \text{transmit}(X T \mid P), c \text{ failed} \]

Expressing Workflows in a Process Calculus

(or finding a good representation if one does not already exist)

Work in progress...
What Are Workflows?

- “A business process is a collection of interrelated work tasks, initiated in response to an event, that achieves a specific result for the customer of the process.”
  -- Sharp and McDermott

- A tendency to consider workflows a subset of business processes

- But I have not yet found (or needed) a workable, rigorous distinction.

Why Formal Workflows?

- Ad hoc workflow handling:
  - Error-prone (no guidance)
  - Significant training needed
  - Lends itself badly to analysis and change

- Formal representation of workflows:
  - Alleviate the above
  - Execution model (computers can execute workflows)
  - Static and runtime consistency checks
  - Lends itself well to outsourcing, service-oriented architecture, and partial automation
Why Formal Workflows?

- Users
  - can easily adhere to established best practice
  - know what tasks can be dealt with now/later
  - receive help to delegate tasks appropriately
  - need only local knowledge about the tasks they solve
    (as opposed to global knowledge about the entire workflow)

- Designers/planners
  - can more easily map out and change processes
  - can introduce structure along the way (ad hoc)
  - can perform formal analysis on workflows
  - can partially automate outsourcing etc.

- Controllers
  - gain finer registration of resource consumption (e.g., time) and thus costs (get Activity-Based Costing for free)
  - can carry out performance analysis more easily

..but this is Taylor’s Scientific Management all over again!

- Workflows can be as freeform or as rigid as one designs them.
- The simplest workflow is just a completely unstructured task list.
- Workflow can be adaptive, starting with the completely unstructured workflow first and imposing structured constraints over time.
Do Workflows Have A Semantics?

- No widely accepted semantics so far.

- Studies by Aalst et al. proposed:
  - 20 control-flow patterns (parallel, sequence, choice, repetition etc.)
  - 39 data patterns (scope, call-by regime, parameters etc.)
  - 43 resource patterns (delegation) (implicit, queue, role, case, quality check)

- The patterns are:
  - imprecise (no clear semantics)
  - overlapping (patterns overlap in non-obvious ways)
  - too inclusive (maybe "goto"-type patterns should be eliminated)
  - non-exhaustive (easy to think of more patterns, but a proper language should be defined instead)

Today's topic

The 20 Control-Flow Patterns

<table>
<thead>
<tr>
<th>Basic Control Patterns</th>
<th>Patterns Involving Multiple Instances</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Sequence</td>
<td>12 MI without synchronization</td>
</tr>
<tr>
<td>2 Parallel Split</td>
<td>13 MI with a priori known design time knowledge</td>
</tr>
<tr>
<td>3 Synchronization</td>
<td>14 MI with a priori known runtime knowledge</td>
</tr>
<tr>
<td>4 Exclusive Choice</td>
<td>15 MI with no a priori runtime knowledge</td>
</tr>
<tr>
<td>5 Simple Merge</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Structural Patterns</th>
<th>Cancellation Patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 Multiple Choice</td>
<td>10 Arbitrary Cycles</td>
</tr>
<tr>
<td>7 Synchronizing Merge</td>
<td>11 Implicit Termination</td>
</tr>
<tr>
<td>8 Multiple Merge</td>
<td>19 Cancel Activity</td>
</tr>
<tr>
<td>8a N-out-of-M Merge (new)</td>
<td>16 Deferred Choice</td>
</tr>
<tr>
<td>9 Discriminator</td>
<td>16a Deferred Multiple Choice (new)</td>
</tr>
<tr>
<td>9a N-out-of-M Join</td>
<td>17 Interleaved Parallel Routing</td>
</tr>
<tr>
<td></td>
<td>18 Milestone</td>
</tr>
</tbody>
</table>
Some Patterns

**Pattern 1: Sequence**
Execute activities in sequence.

**Pattern 6: Multiple Choice**
Choose several execution paths from many alternatives.

**Pattern 13: Multiple Instances**
Generate many instances of one activity with synchronization.

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

More Patterns

**Pattern 10: Arbitrary Cycles**
Execute workflow graph without any structural restriction on loops.

**Pattern 20: Cancel Case**
Cancel (disable) the process.

**Pattern 16: Deferred Choice**
Execute one of two alternative threads. The choice should be implicit.
Calculus of Communicating Systems

- CCS syntax:

\[
P ::= 0 | \tau.P | a?x.P | a!x.P | P + P | (P | P) | \text{new} a \ P | a?^*x.P
\]

Atomic tasks are simply written as \(a, b, c\) etc.

- **Pattern 1: Sequence**
  - \(a.P\) – not general enough
  - \(P.Q\) – not syntactic
  - Require processes to signal termination explicitly on a designated channel, e.g. \(\text{ok}\)
  - new \(\text{go}\) (\(P \ [\text{go/ok}] \ | \ \text{go?} . Q\))

Example Encoding

- **Pattern 7/9: Multiple Choice/Discriminator**
  - \((\tau.P_1 + \tau.\text{skip}) \ | \ ... \ | (\tau.P_n + \tau.\text{skip})\) | \(\text{ok?} . (Q \ | \ \text{skip?}^* . 0 \ | \ \text{ok?}^* . 0)\)

\[
\begin{array}{c}
\text{Mult} \\
\rightarrow \\
\text{Disc} \\
\rightarrow \\
\text{Q} \\
\rightarrow \\
\text{P_1} \\
\rightarrow \\
\cdots \\
\rightarrow \\
\text{P_n}
\end{array}
\]
Example Encoding

- **Pattern 10: Arbitrary Cycles**
  - new go_a, go_b,
    
    \[\text{a.go}_a! | \text{go}_b? \cdot \text{b.go}_c! | \text{go}_c? \cdot \text{c.(go}_b! + \text{d.(go}_c! + \text{e})}\]

- **What if Merge was Sync. Merge?**

Observations on the patterns

- **Split and join are separate / Free structure**
  Patterns have graph-type structure (YAWL, Petri nets, WFDL) as opposed to block structure (XLANG, parts of BPEL)

- **Deferred vs. explicit choice**
  Branching time semantics may be needed

- **Many split and join variants seem better expressed by simple data-language**

- **Cancellation**
  Neither Petri nets nor pi-calculus handle this very well. LOTOS and CSP have operators for this.

- **Multiple Instances needed**
  Generativity essential
Synch. Merge Has Non-Local Semantics

- Workarounds
  1. Integrate with split-end pattern (thus loosing free-form property)
  2. Decide runtime (thus loosing some static checks)
  3. Pass around meta-information from previous split(s) (thus complicating distribution and introducing complexity, also insufficient in degenerate cases)

More Issues

- Semantics of Discriminator is unclear (but can be well-defined)
- Can tasks be cancelled? (sure, tasks can be processes too!)
- Multiple Instances combine several other patterns (split, join, and repetition)
- Synch, Merge and Sequence overlap somewhat subtly
A Formal Semantics for the Patterns?

- More precise benchmarking of workflow languages
- Should be model independent (not Petri net, \(\pi\), EPC)
- Should split and join be separated or not? (and how about *Multiple Instances*?)
- Should all patterns be covered? Directly or by data-manipulation language?
- How to compare expressiveness? Felleisen-style?

SMAWL – a SMALL Workflow Language

- Preliminary language design exercise
- Design criteria:
  - Cover all 20 patterns (and more!)
  - Minimize need for synchronization primitives
  - Retain strong link to \(\pi/\text{CCS}\)
  - Independent of data patterns
    (Allows orthogonal data manipulation language to be plugged in.)
- Method
  - Collect patterns in a few, general constructions
  - Describe a source-code transformation to \(\pi/\text{CCS}\)
A Workflow in SMAWL

workflow Q =
  Intro;
  choose any {
    ⇒ Workflowmønstre;
    Kodning;
    SMAWL
    ⇒ Kontrakter;
    CDL
    ⇒ REA
  }
};
Arkitektur;
Afslutning
end

SMAWL Syntax

W ::= workflow w = P end
D ::= fun f = P end D | newlock (l, u) D
    | milestone(ison, isoff, sel, clear) D | ε
P ::= activity | send(f) | receive(f) | call(f) | P; P | lock(l, u){P}
    | choose any (wait for k){PP merge(n) P} | choose one{PP} | cancel {P}
    | do all (wait for k){PP merge(n) P} | multi{n}{P}
PP ::= ⇒ ρ P PP | P PP | ⇒ P
Britney Wants to Sing

\textbf{workflow} Become a recording star =
\begin{verbatim}
chooseone {
    \rightarrow \text{call \{Work your way up\}}
    \Rightarrow \text{call \{Try to get lucky\}}
}\}:
Make record;
doall {
    \Rightarrow \text{chooseone \{}
    \Rightarrow \text{Develop as an artist}
    \rightarrow \text{Develop bad habits}
    \}
    \Rightarrow \text{Rehearse tour;}
    \text{Do tour}
\}
\end{verbatim}

Transforming SMAWL to CCS

- Sequence
\[ T[P; Q] = \lambda ok. \text{let } ok' \Leftarrow \nu() \text{ in } T[P]ok' | ok'? . T[Q]ok \]

- Multiple Instances
\[ T[\text{multi}(n) \{ P \}] = \lambda ok. \text{let } create \Leftarrow \nu() \text{ in } \text{let } ok' \Leftarrow \nu() \text{ in } \\
\text{create!} . \ldots . \text{create!} . ok' . \ldots . ok' . ok! | \text{create?}* . T[P]ok' \]
Future Research on Workflows

- Set up success criteria for process model and data-manipulation language
- Formal semantics for workflows
- Consider distributed workflows
- Find/adapt/design suitable process calculus
- Implement a workflow language

A Process-Oriented System Architecture

(Highly speculative)
Future: A Process-Oriented Framework

- Process language
- Base (data) language
- Service patterns
- Auto. GUI-generation

- Data model
- Process mining
- Inter-operability (ontology)

Ongoing NEXT Projects

- Finite Differencing for Realtime Reporting
- REA Bookkeeping
  (Double-entry bookkeeping is not the only option)
- Plan X: Value-Based Programming
Related Work

- **Concurrency models**
  - Process calculi (CCS, pi, CSP, Bigraphs)
  - Petri nets

- **BPEL – Business Process Execution Language**

- **YAWL – Yet Another Workflow Language**

- **WS-CDL - Choreography Description Language**
  In particular, see Kohei Honda’s recent stuff on behavioral type systems.

- **Transactors: a programming model for maintaining globally consistent distributed state in unreliable environments**
  John Field and Carlos Varela, POPL ’05

- **PiDuce – a project for experimenting with web services technologies**
  Samuele Carpineti, Cosimo Laneve, Luca Padovani, Jan. 2006

- **Compositional Contracts**
  Peyton Jones, Eber, Seward, 2001

Q&A and thanks!

- Thank you for hosting me!
- Questions?

- More info on:
  - http://www.stefansen.dk
  - http://www.it.edu/next/