A Declarative Framework for Enterprise Information Systems

Qualification Report

Christian Stefansen

August 22, 2005

In memoriam

Taras Gapotchenko

Chris Bartok

Summary

The rapid spread of the internet in the last decade lead to an increasing interest in frameworks for supporting business process automation across organizational boundaries. In the wake of this change, a host of standards have been proposed, all claiming to support this directly or at least provide the architectural scaffolding. No standard has yet gained academic recognition and widespread use in industry but recent years have seen a convergence of interests from several areas: System vendors, workflow researchers, and standards bodies are now all focused on building models for distributed, communicating, and, possibly, mobile business processes. Ideally, such a model should be brought to bear on the substantial body of research in currency models as well as the plentiful streams of commentaries associated with industry standards. While such a marriage has not occurred yet, it seems to be in the making.

We approached the challenge of building such a business programming model from three directions of research:

1. **A contract/choreography language.** An choreography language takes a perspective independent of any particular of the parties involved in a structured exchange. Inspired by Peyton Jones' seminal paper on Compositional Financial Contracts, a language for commercial contracts is designed. The contract engine is a reactive system: when notified that an event has occurred, the contracts are rewritten according to the rules of the reduction semantics (or a breach of contract exception is raised). The contracts are compositional and moreover the result of any reduction has a syntactical representation in the contract language. This key feature renders the contracts amenable to analysis at any point of reduction. As it turns out, the contract language has a serendipitous similarity to the Web Services Choreography Description Language (WS-CDL). We briefly discuss this and a projection to map a multi-partner contract to a description of the protocol (i.e. obligations and rights) of a single partner. This projected description serves as the skeleton for each participating partner, and using their business process or workflow language, they can fill out the details describing how to fulfill their end of the contract.
2. **A workflow language.** While most workflow products and researchers have used Petri nets as the underlying concurrency model, there has been a push towards algebraic methods (for purposes of reasoning) and models that provide first-class primitives for mobility (i.e. channel-passing, locality, and process-passing). Today, the *de facto* benchmark for workflow models is a large number of *workflow patterns* (*control-flow patterns*, *dataflow patterns*, *interaction pat-*

terns, and *resource patterns*) compiled by van der Aalst. We demonstrate that all control-flow patterns can be expressed in a process calculus (Calculus of Communicating Systems, CCS) and subject the patterns to a closer analysis. We then consider a taxonomy of workflow languages and discuss some important languages with respect to this taxonomy. Last, we present a small workflow language that aims to accommodate all control-flow patterns while maintaining a strict formal semantics through its direct connection to CCS.

3. **An alternative accounting model.** We consider to accounting models: The all-time classic, *double entry bookkeeping*, in formal clothing and the heretical *Resources/Events/Agents* (REA) accounting model. We claim that an enterprise system can be built from first principles based on any of these models and a language for business processes.

The research on the contract language and the REA model has already been thoroughly described and published. The main effort is therefore on the research pertaining to workflow languages where only a small paper has been published so far. This three-pronged research comes together to form a process-based enterprise system architecture. The architecture is outlined in the last chapter of the report along with directions for future research.

Contents

1 Introduction **7**
- 1.1 A Brief History of Accounting 7
- 1.2 Contributions 10
- 1.3 Overview 10

2 Commercial Contracts **12**

3 Expressing Workflows **16**
- 3.1 Definition 16
- 3.2 Why Express Workflows Formally? 17
- 3.3 Design Criteria, Requirements, and Patterns 20
- 3.4 Workflows as Programs 21
- 3.5 Control Patterns 23
- 3.6 Data, Resource, and Service Interaction Patterns 50

4 A Model for Business Data **52**
- 4.1 Double-Entry Bookkeeping 52
- 4.2 Resources/Events/Agents (REA) 55
- 4.3 Conclusions and Future Work 58

5 Toward a Declarative Framework **60**
- 5.1 Architecture 61
- 5.2 Requirements, Standards and Models 62
- 5.3 Business Rules and Verification 63

6 Conclusions and Future Work **64**

A Publications **65**

Preface

The first two years of my Ph.D. program have resulted in three reviewed paper publications, two technical reports, one poster, one position paper, a number of talks, and a few other contributions. A complete list can be found in Appendix A. This report is based in part on those writings and in part on hitherto unpublished work.

During the first year of the program I was physically located at DIKU at the University of Copenhagen whilst collaborating with ITU/NEXT affiliates on a weekly basis. The second year of the program I spent at Harvard University under the mental guidance of professor Greg Morrisett.

This is a status report. It is a basis for a discussion, it represents unfinished work, it is speculative and hopefully this means fun. It will be interesting to debate viable avenues of research in the years to come. For this reason, a substantial number of flimsy ideas for future work are left in the report. I have made no attempts to limit these.

Acknowledgments

I would like to warmly thank Microsoft Business Solutions for having sponsored my Ph.D. through the NEXT project and for being a flexible, committed, and colloquial research partner all the way. Thanks to Fritz Henglein, my dedicated and very energetic advisor. Thanks to Greg Morrisett for taking in a random kid from Denmark. My co-authors who have agonized and laughed with me: Jesper Andersen, Signe Ellegård Borch, Ebbe Elsborg, Jakob Grue Simonsen. The funky NEXT folks from MBS, DIKU, and ITU. My office mates at Harvard University, Amal and my great friend Peter Pietzuch. And all those nice MD people and DIKU people who are too numerous to list.

I remain humbly grateful to the Fulbright Commission for supporting me and making this happen. Also thanks to Anton Hansen & Hustrus Mindelegat for its continuing support. Thanks to Jacob Riff for running our company in my absence (are we bankrupt yet?) and to Karin Outzen for being a great and indispensable help in so many ways. Thank you, my dear Cambridge roommates, Maria, Taras, Kim, and Jess, for taking good care of me when I was bogged down by work. Thanks to Espen for starting everything by referencing me, Morten and Tommy for gentle day-to-day mockery, and so many other people (I'll buy you a beer if I forgot you). Thanks also to Heidi for a tremendous amount of support.

Lastly, thanks to Jorge Cham for his comic PhD (Piled Higher and Deeper), which helps graduate students throughout the world appreciate that they are not alone, and to the Economist for providing a short and concise style guide (which I would read more attentively, had I the time).

Chapter 1

Introduction

The main objective of this work is to design:

> *A declarative programming platform for writing enterprise systems and automating business processes in a service-oriented architecture.*

Before scrutinizing the question further, let us trace some of the main developments that lead to the current state of affairs.

1.1 A Brief History of Accounting

Throughout the history of computers, business systems have always been among the first to establish a foothold in industry when new technological advances were made. The financial sector was among the first to utilize mainframe computers for critical business applications, and with the introduction of the personal computer pioneered by IBM in the 80s, software packages for small business financials[1] were among the first to reach a broader market. The foundation of all financial systems was the principles of double-entry bookkeeping, a system employed by merchants of Venice and Naples in the 13th century. The system was described in 1458 by the merchant Benedetto Cotrugli in *Il libro dell'arte di mercatura*, and the first printed account was given by Luca Pacioli in 1494 in *Summa de Arithmetica, Geometria, Proportioni et Proportionalita*. The system subsequently became the basis of all accounting in the Western world.

Double-entry bookkeeping in its pure form, however, handles only financial accounting. Even before computers entered the realm of accounting, one had to resort to auxiliary systems outside the ledger. There were paper-based systems such as customer address books, inventory lists, payroll records, and contracts. In the days before computers the development of such ad hoc systems did not matter much because ideas such as interoperability and distribution were only theoretical. When the first systems were developed, they too—for lack of viable alternatives—incorporated ad hoc auxiliary systems. There was some good to be said about this, namely that the systems were strictly modular and maintained well-defined boundaries between their sub-systems.

[1] Such systems handle accounting, accounts payable, accounts receivable, inventory, and possibly more.

Furthermore, there would only be one-way updates between the sub-systems (usually the ledger would be the bottom element), and hence no spurious cyclic updates would occur.

The 80s saw a significant paradigm shift as Harvard Business School's business strategy icon, Michael E. Porter, agitated for a focus on the processes by which the company generates added value. The shift from data-centric to process-centric management became apparent in the models and business lingo: Value Chain, Activity-based Costing, Business Process Re-engineering, Outsourcing, Supply-Chain Management, Core Business etc. [7]. In essence, these regard the enterprise as a chain of processes designed with the purpose of generating added value to its customer from input to its output. If some enterprises, processes or parts of processes do not generate added value, they should be trimmed or they will be out-competed, the theory goes[2]. Total Quality Management—a reincarnation of Taylorism/scientific management that had fallen from grace earlier—entered the stage, prescribing measurement and continuous process improvement.

In addition to changing strategic thinking, the process perspective changed accounting from a supporting function to a management tool. The shortcomings of current accounting had been apparent for a while, and a host of ideas under the general label of *management accounting* emerged. In this vein, Robert Kaplan and Robin Cooper proposed a new cost allocation method, *Activity-based Costing*. In activity-based costing one allocates costs to products and services, rather than to traditional functional divisions of the enterprise. This makes apparent the cost of maintaining a product line, not just in actual production, but throughout the entire enterprise from requisition to post-sale support—it gives an approximate allocation of costs to business processes. Activity-based costing—not a profound idea on the surface of things—was made possible in part by ERP systems, which allowed a much finer-grained registration of business activity than what had been possible previously.

Another management rave was that of Just-In-Time management (JIT). Toyota Motor Corporation, the Japanese car manufacturer, sought ways of reducing the capital tied up in inventory at any given time. To this end, parts would be produced not to stock, but just in time to be finished for their immediate assembly and delivery. Just-In-Time took on a wider meaning as enterprises extended the idea to suppliers as well; an idea known as supply chain integration[3]. Not only would the enterprise' own production be

[2] In a textbook example, Nike, Inc. does not produce sneakers itself. Nike adds value through its brand, design, and quality control; the rest is outsourced because Nike then can choose a number of competitive manufacturers in the marketplace and maintain flexibility.

[3] Supply Chain Management (SCM) is a somewhat vague notion that sometimes seems to take on ad hoc meaning in certain contexts. Researchers and practitioners have attempted to define SCM in a number of different ways (cf. [7]). Here are two definitions:

> **The Definition of Logistics** Logistics management is that part of the Supply Chain Management process that plans, implements, and controls the efficient, effective forward and reverse flow and storage of goods, services, and related information between the point of origin and the point of consumption in order to meet customers' requirements. *(Council of Supply Chain Management Professionals,* `http://www.cscmp.org/`*)*
>
> **The Definition of Supply Chain Management** Supply Chain Management encompasses the planning and management of all activities involved in sourcing and procurement, conversion, and all Logistics Management activities. Importantly, it also includes coordination and

just in time, so would (recursively) ordering parts from suppliers. The impact was clear in the ERP market: enterprises needed systems that would integrate with the inventories and order systems of their suppliers—and with the procurement systems of their customers. Fluctuations in market demand, one hoped, would automatically propagate back through the supply-chain to avoid excessive stock-piling or supply shortages.

The emphasis on outsourcing, supply chain management, and business processes applied pressure on current ERP systems: They needed a higher degree of interoperability, and they needed to replace the division into functional silos[4] with a process-oriented model. In the first iteration, these observations lead to the very recent surge of interest in interoperable data formats (e.g. XML), automating business processes over the Internet (cf. OASIS, WS-BPEL), and *Service-oriented Architecture* (SOA). These ideas all sit on the rim of the enterprise and provide an integration front end to the ERP systems. The ERP systems sitting behind are still, by and large, remnants of a time when companies where thought of as functional silos. To be fair there is a good integration between the modules, but the approach to business process programming has been to add new orchestration layers on top of existing code. More suggestively, a lot of financial controller work today goes into importing, manipulating, and maintaining data in *ad hoc* systems (Excel spreadsheets, text files). Such auxiliary management information systems are symptomatic for an ERP system that does not store/provide information in the desired formats.

In the second iteration, it is time for enterprise systems to embrace a process perspective that does not only sit on the edge, but permeates the system—in the way it is written, maintained, and used. A system that does not distinguish between program and business process; a system allowing processes to span organizational boundaries and migrate; and a system that allows reporting directly on processes without convoluted *ex post* cost allocation schemes.

We believe the time is now.

1.1.1 Current Products

The last few years have seen a concentration in the market for ERP systems. JD Edwards was bought by PeopleSoft, which in turn was bought by Oracle Applications after a long and painful series of hostile take-over attempts. Microsoft Business Solutions bought Great Plains and Navision, which had just completed a merger with Damgaard Data, leaving PPU Maconomy as one of the only other significant player in the Danish market. Another major player, SAP maintains a solid position in the large-scale manufacturing segment worldwide, and other actors include SSA Global, which acquired BAAN, and the United Kingdom-based Sage Group plc. A few open source systems have appeared, most notably Compiere[5], but their impact remains negligible.

> collaboration with channel partners, which can be suppliers, intermediaries, third-party service providers, and customers. In essence, Supply Chain Management integrates supply and demand management within and across companies. *(Council of Supply Chain Management Professionals,* `http://www.cscmp.org/`*)*

[4] E.g. human resources, sales, procurement, manufacturing, finance.

[5] http://www.compiere.org

From a research point of view, though, they come in very handy for benchmarking and experimentation purposes.

Today's Enterprise Information Systems typically have more than a million lines of code, use more than 500 database tables, contain terabytes of data, and with the rise in Internet commerce service more than 10000 clients simultaneously. Enterprise Information Systems today continue to be the most heavyweight systems one is likely to encounter in industry.

1.2 Contributions

As stated we seek to design a declarative programming platform for writing enterprise systems and automating business processes in a service-oriented architecture. The contributions in this report are:

Chapter 2 See the technical report and article [4, 5].

Chapter 3

1. A CCS encoding of the 20 control flow patterns in Table 3.1
2. An 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. A presentation of a speculative *overlay* operator and its use for certain workflow scenarios
4. A short discussion of using CCS vs. π-calculus for workflow description languages.
5. A further analysis of control patterns and directions for future research.

Chapter 4

1. See the article [9]
2. A formalization of double-entry bookkeeping

Chapter 5 A sketch of a speculative architecture for process-oriented enterprise systems bringing together the contributions of the previous chapters.

1.3 Overview

It is convenient to make a coarse distinction between three interconnected aspects of Enterprise Information Systems:

1. *Business Data*
2. *Business Processes/Rules*
3. *Business Reporting/Monitoring*

Business Data deals with the data models and the metadata to support security enforcement. While this is an aspect of great practical importance, this report will make do with a look at the core *Resources/Events/Agents* model *(*REA) and the double-entry ledger.

The *Business Process* aspect is concerned with the dynamic nature of Enterprise Information Systems: How do processes behave operationally? What underlying concurrency model is appropriate and what properties should it guarantee? How are rules enforced statically and dynamically on processes? Can processes be model checked or proven correct? How can analyses be made on running programs? What information flow guarantees are needed and provided?

Business Reporting/Monitoring deals with the issues involved in the efficient reporting and monitoring over a very large log of events and running processes.

This report focuses on (2) Business Processes/Rules, with the exception of a small detour into the realm of Business Data where the Resource/Events/Agents accounting model (REA) is discussed. First, we visit two different areas for concurrency models in business applications. Based on these experiences, the last chapter suggests a design for a general declarative business programming platform.

Commercial Contracts Chapter 2 summarizes previous work on how to build a system for modeling, executing, monitoring, and analyzing commercial contracts. This language turns out to have an interesting relation to choreography languages such as WS-CDL.

Expressing Workflows Chapter 3 describes how to express workflows in CCS and discusses the shortcomings of the current *de facto* control flow patterns for workflows. It introduces SMAWL, a CCS extension for workflows, and identifies directions of further research.

A Model for Business Data Chapter 4 presents two candidate data models. One is the double-entry ledger, the other is the REA accounting model. We claim that a complete financial system can be based on a small process language and one of these data models in lieu of using a generic language.

Toward a Declarative Framework Chapter 5 discusses how to make the contributions of the three previous chapters come together to form a framework for business programming. A very speculative system architecture is sketched.

Chapter 2

Commercial Contracts

Commercial contracts are contracts governing the exchange of goods, services, and rights in the marketplace. Contracts contain descriptive clauses qualifying the nature of the scarce resources in question and prescriptive clauses explaining obligations, rights, and restrictions on the involved parties and the temporal and logical structure of these. Today, contract handling is, by and large, manual and informal, and this precludes a number of interesting possibilities, namely monitoring, execution, and analysis. Inspired by the seminal paper on Financial Engineering by Peyton Jones [32], our research seeks a way of representing the prescriptive component of contracts formally. Commercial contracts have been covered extensively in previous work [4, 5], so this chapter will make do with a short example and some directions for future research.

Consider figure 2.1 for an example. This contract makes clear the need for choice, parallelism, sequence, and repetition. These are the four most basic control-flow patterns. They give rise to the syntax shown in figure 2.2. Using this syntax the agreement to provide legal services can now be expressed as shown in figure 2.3. The contracts react to observed events in a reduction semantics. The most important property of the language is that for the large class of *guarded* contracts, there always exists a syntactical representation of a contract after every reduction step. This means contracts can be represented as syntactical objects at any point during execution—there is no need to resort to e.g. trace sets or auxiliary state information.

The commercial contracts take a global perspective. That is, they describe only observable communication *between* the parties of the contract. Internal processing is

Figure 2.1 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 receiving an invoice.

Section 3. This contract is valid 1/1-12/31, 2005.

Figure 2.2 Syntax for contract descriptions

Success/Failure

The succeeded/failed contract with no commitments.

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).

$c_1; c_1$

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.

$f(\vec{a})$

Expansion to body of contract f with arguments $\vec{a}$.

letrec $f_i[\vec{X_i}] = c_i$ **in** c

Contract c with named contracts f_i with formal arguments X_i bound to c_i.

Figure 2.3 Code for the agreement to provide legal services

```
letrec
 legal (att, comp, hours, payment, extraprice, end, n) =
     transmit (att, comp, H, T | n <= T and T < n + 30 d and T <= end).
      (transmit (att, comp, invoice, T1 | hours < #(H, number, n + 30d)
       and #(invoice,total,T1) = (#(H,number,n) - hours) * extraprice).
       transmit(comp, att, #(invoice,total,T1), T2 | T2 <= T1+45)
       + Success)
   || (legal (att, comp, hours, payment, extraprice, end, n + 30 d)
       + Success)
  || transmit (comp, att, payment, T | T <= n + 40 d)
in legal ("Attorney", "Company", 20, 10000, 1200, 2004.12.31, 2004.1.1)
```

not dealt with. This is fundamentally different from the perspective used in the next chapter, where we take the *local perspective* or the *trading-partner perspective*. In the local perspective we are interested in only one party, who send communication signals and executes workflows in reaction to inbound communication. Given a contract describing the global perspective, every party can compute his *projection* and use this as a skeleton for his local process design. Afterward it can be checked if the local business processes conform to the global contract. This is a promising direction of future work.

Our language of commercial contracts has most of the basic control flow constructs in common with WS-CDL [33], the choreography language from OASIS. In CDL the *ordering structure* is governed by four constructs: *sequence*, *parallel*, *choice*, and *perform activity*, where perform activity corresponds to our function invocation. Atomic activities include *interaction activities* (our *transmit*), *assign* (embedded in *transmit*), and *no action* (Success), but also *silent actions*, which do not compare directly to our language. In addition CDL has an elaborate structure on *roles*, *relations*, and *participants*, as well as explicit mobile channels (linear or unlimited) and exceptions/finalizers. Predicates are referred to as *guards,* and contrary to our contract language, guards enclose entire blocks rather than sit directly on a *transmit*. CDL, like our contract language, is parametrized over the base language for arithmetic expressions (XPath is the *de facto* base language). Observables—i.e. environment variables such as stock prices, exchange rates etc.—are provided as XPath extension functions directly accessible to the base language. In light of the striking similarities, several research directions seem interesting. First, the semantics of our contract language could be extended to handle CDL and thus provide a succinct formal semantics to CDL. Second, conformance checking between CDL/our contract language and BPEL/our workflow language would be valuable. Third, section 3.5 deals in detail with control flow patterns; the expressiveness of CDL could investigated by a patterns-based analysis. Possibly, several choreography patterns would result from this.

The contract language in isolation also poses interesting questions. First, a syntactic brush-up would be welcome, explicit (deterministic, immediate) branching using if/then/else would be convenient, as would a normal form of pattern matching à la Standard ML meets Join calculus. A form such as

$$\begin{array}{rcl} (event(s), time) & \Longrightarrow & C_1 \\ \mid (event(s), time) & \Longrightarrow & C_2 \\ & \cdots & \\ \mid (event(s), time) & \Longrightarrow & C_n \end{array}$$

would aid programming and presentation a lot. The major selling point of a domain-specific language like this is that of analyzability, programs double as data. To drive this point home, more analyses are needed. To this end, it should be seriously considered if full recursion is really needed. The current language allows full recursion, and at the same time preliminary studies [47] have shown that even with a very simple contract language, analysis is surprisingly involved. More effort must go into closing this gap in a satisfactory way. Most contracts we have encountered so far can be statically unfolded based on a number of unfolds or a date, thus eliminating the need for full recursion.

As a last direction of research the contract language could reclaim its relationship with the derivative market. The valuation of compositional contracts would be immensely useful in these markets as new derivatives are designed on a regular basis. A reasonable first milestone is to recover the Black-Scholes equation for European option pricing in a non-arbitrage market. It is not clear if compositional analysis alone can do this, since this requires the Black-Scholes equation to be expressed as a linear combination of the strike price and the current stock price.

Chapter 3

Expressing Workflows

In this chapter we will look a workflows and how to express them in a way that is representable in a computer. First workflows are defined (Section 3.1), then we consider the problems of having a loose or no workflow handling (Section 3.2). This leads to the design criteria for a workflow model. We then take a small detour and compare workflows with conventional programs to establish an understanding of the differences (Section 3.4). Afterward, a workflow model based on Calculus of Communicating Systems is presented and evaluated with respect to the design criteria (Section 3.5).

The text is interspersed with comments on two of the most important process languages available: YAWL and BPEL. These two languages alone cover a very broad spectrum of workflow issues. YAWL [67] is an academically developed workflow language, which is probably more expressive than any of the industry offerings, to which it has been compared. BPEL [65] is the industry flagship standard for business process execution. There are many other standards and products. For an overview see van der Aalst [71] and p. 172 [72].

3.1 Definition

"A formal definition in plain English" far too often becomes an oxymoron. Two important textbooks on workflow modeling venture to give definitions, and these are quoted below to convey some amount of intuition:

> "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 [60]

> "A workflow comprises cases, resources, and triggers that relate to a particular process" and "[t]he definition of a process indicates which tasks must be performed—and in what order—to successfully complete a case. In other words all possible routes are mapped out. A process consists of tasks, conditions, and subprocesses. By using AND-splits, AND-joins, OR-splits, and OR-joins, parallel and alternative flows can be defined.

> Subprocesses also consist of tasks, conditions, and possible further subprocesses. The use of subprocesses can enable the hierarchical structuring of complex processes." — van der Aalst and van Hee [72]

Wikipedia [78] provides this definition:

> "Workflow is the operational aspect of a work procedure: how tasks are structured, who performs them, what their relative order is, how they are synchronized, how information flows to support the tasks and how tasks are being tracked. As the dimension of time is considered in Workflow, Workflow considers 'throughput' as a distinct measure. Workflow problems can be modeled and analyzed using Petri nets."

Readers interested in a Petri net-based approach are referred to van der Aalst and van Hee's "Workflow Management" [70]. Sharp and McDermott give a treatise with a focus on the business aspects of workflow modeling in "Workflow Modeling" [60], which also ties in with the managerial vogues of the late 80s and 90s such as TQM[1], Kaizen, MBO[2], and MBWA[3].

3.2 Why Express Workflows Formally?

The simplest conceivable workflow system would simply consist of task list, where tasks would be checked off when completed. To complete the workflow the user would either draw on domain knowledge to figure out what tasks should precede others or proceed by trial and error, suspending tasks until the necessary preconditions were satisfied. This approach works in very simple scenarios such as personal task lists but fails to scale to larger workflows with more agents:

- **No aid in adhering to established practice.** If an established best practice exists, the system has no way of helping the user follow this. Designing and accounting for best practices is becoming increasingly important with the introduction of legislative auditing requirements such as GAAP (Generally Accepted Accounting Practices) and the Sarbanes-Oxley Act[4].
- **No simple/automatic way of deciding what to do next.** The interdependence between tasks is not represented and so the required task- and data-dependency analysis must be brought to bear solely on the user's domain knowledge in an *ad*

[1] Total Quality Management

[2] Management By Objectives

[3] Management By Walking Around

[4] Officially known as the "Public Company Accounting Reform and Investor Protection Act of 2002". In particular the aptly named section 404(a) has an impact on internal business processes in stating:

> (a) Rules Required. The Commission shall prescribe rules requiring each annual report required by section 13(a) or 15(d) of the Securities Exchange Act of 1934 to contain an internal control report, which shall–
> (1) state the responsibility of management for establishing and maintaining an adequate internal control structure and procedures for financial reporting; and
> (2) contain an assessment, as of the end of the most recent fiscal year of the issuer, of the effectiveness of the internal control structure and procedures of the issuer for financial reporting.

hoc fashion. This makes it more difficult to delegate and divide labor—especially to new employees with limited training.

- **No way to express new tasks created as result of other tasks' outcomes, repeating tasks or choice structure.** Again, all this information must be learned, maintained, and remembered by the users.
- **No simple/automatic way of deciding how to best delegate tasks at hand.** Delegation can be based on e.g. experience/skills, current workload, expected processing time, confidentiality, one handler pr. client, or even internal auctioning[5].
- **Training users is more costly.** Very little or no domain knowledge is explicitly represented so proper use relies heavily on correct user understanding of the entire workflow.
- **Changing business processes is costly and potentially resultless.** In the absence of a system providing guidance and support, users may revert to old practices. Changing processes is costly, because the current processes must first be mapped through interviews, and users must be taught the new processes through practice and repetition rather than by system guidance.

Since some knowledge of the structural constraints of the tasks is needed in any case to perform a number of routine analyses, it seems reasonable to suggest that these constraints are captured explicitly in the workflow system. Some additional benefits of an explicit representation of workflows are:

- **Formal analysis.** This foremost argument for domain-specific languages is this: Programs in a DSL double as data that can be analyzed and transformed. This means that a workflow written in a DSL for workflows immediately lends itself to analyses, pertaining to e.g. reachability, forward/backward slicing, deadlock, longest/shortest propagation delays, resources needed, scheduling/planning etc. This analysis can be carried out statically on a program, but also on a running program. Languages with reduction semantics—i.e. an execution semantics where every atomic step in the execution can be represented as a program within that language—are particularly convenient to analyze, because the state is represented explicitly in the program expression. Likewise, a language that is built up on several orthogonal components (control flow, data flow etc.) is more likely to allow easy access to *projection*, that is, zooming in on one particular aspect of the program, abstracting away from everything else—for purposes of development, analysis or scrutiny.
- **Redesign aid: longest propagation delay, dependency** Formal analysis is not only useful for timely business reporting, but also for redesign. Workflow design tools can help identify long paths, resource-demanding sections etc. in the design process. Thus workflow design tools become an aid for process redesign.
- **Activity-based costing.** In a workflow-driven system, costs are immediately available pr. employee, pr. case, pr. client, pr. tasks type etc. And this is without

[5]No other works to our knowledge have mentioned this idea. It is an interesting way of distributing bonuses to employees who volunteer for particularly demanding tasks. Game theoretically, of course, this should be a Vickrey auction or a similar mechanism where truth-revelation is a dominant-strategy equilibrium (cf. `http://en.wikipedia.org/wiki/Dominant_strategy`).

the extra work of assigning cost dimensions on job scheduling entries as seen in current ERP systems.

- **Performance analysis pr. employee, process, task etc.** Very similar to the above. Performance analysis potentially becomes more fine-grained because all resources spent are actively mapped to known processes.
- **Design time or even runtime flexibility trade-off.** No one desires an overly rigid workflow system—and no one knows exactly at design time what will be too rigid and what will not. Workflow systems can be normative (allowing the users to skip and jump tasks and their leisure), prescriptive (forcing the user to follow the workflow) or anything in between. The important observation is that this decision does not have to take place design time. Instead the system could take a flexible stance allowing all violations in the beginning and imposing stronger rules along the way. The trace of events will provide statistics about the tasks skipped, by whom, how often, etc. This will help diagnose if the workflow is ill-designed or the company has problem adhering to its own best practice.
- **(Long term) Support for outsourcing, commissioning, subcontracting.** Describing workflows in a language with an established semantics is the first step towards being able to outsource workflows or parts of workflows. Naturally, a workflow only documents the skeleton of the work that has to be done, and the work descriptions for every work item must be filled out as well.
- **First step towards mobile processes.** Finding a formal representation of workflows is the first step toward mobile business processes. This opens up an interesting domain of business models for semi-automated process auctioning over the Internet.

Having enumerated some of the potential gains, we should also note some of the costs and issues involved:

- Work items are coupled with other systems and legacy systems. Sometimes data stay within the systems, sometimes it follows the workflows. There is a significant challenge in providing interfaces for all systems so that a workflow management system gains access to the case data it needs. In these days of XML, a solution seems to be within grasp, though.
- Workflows can easily become too rigid. It is imperative that one begins with a flexible systems and narrows it in only after significant experience.
- Workflows remain a small part of business reality. Telephone calls, corridor conversations etc. are left out (enter *Speech Act* research).
- There is a potential gap between who designs the workflows and who uses the workflows. Reducing this gap should be an ongoing effort. Languages and tools should be easy and fun to use.

Weighing the costs and benefits it appears that significant gains are possible with formal representations of workflows. We should carry on and investigate further.

Although as noted a completely unstructured task list is insufficient for the job, it is useful *ab initio* to think of all workflows being unstructured. We then add structure as needed afterwards. Going from simple workflows to structured workflows can be done by design, by discovery or by a hybrid method. Structuring workflows by design

means having domain experts and users impose the constraints deemed necessary *a priori* or at runtime. Structuring workflows by discovery is done by *process mining*: Given an existing event log, one tries to discover invariants and recast the invariants as workflow structure. This is often done using a genetic algorithm that randomly generates a number of workflows, ranks them according to a fitness measure, and refines them iteratively [76].

For the remainder of this section we will concentrate on workflows structured by design.

3.3 Design Criteria, Requirements, and Patterns

In designing a new language a range of considerations must be taken into account. Ultimately, language design is about the convenience of program writing in the language. This is its domain fit, availability of common idioms, precise error messages, good tools, few possibilities to introduce errors, minimal number of artifacts etc. Many design criteria can be read directly out of the goals outlined in the previous section. Since we are designing a domain-specific language, formal analyzability becomes particularly important—it is a major justification for domain-specific languages. This usually means that the language should be as simple as possible—preferably not Turing complete. Compositionality is another desirable, albeit not strictly necessary, feature to facilitate automated analysis. A very intriguing new area of research is that of process mining, i.e. recovering processes from event traces. The language should ideally lend itself well to program rediscovery by genetic algorithms, but this property is very hard to quantify. It does seem, however, to go hand in hand with the idea of a language with few and simple constructs.

Apart from the general design principles, there are some concrete requirements to address. The idea of *patterns* was popularized in Gamma et al.'s seminal book "Design Patterns" [20]. This sparked a maelstrom of patterns in a variety of research areas until some saturation was achieved. The idea was also applied—and quite successfully so—to the workflow domain: In their seminal 2002-2004 work van der Aalst et al. introduced the idea of control-flow patterns (née workflow patterns) [71, 70, 67]. The result was 20 control-flow patterns identified empirically in current workflows, workflow systems, and workflow standards. Since then this body of work has been extended with data patterns, resource patterns, and most recently service interaction patterns. Moreover, the patterns have become the *de facto* standard for workflow systems benchmarking. This means that as a bare minimum, one must take into account the patterns, and for each of them attempt to cover it or deliver a reasonable explanation for its omission. While seemingly straightforward, this process is impaired somewhat by the current state of the patterns. The patterns lack a formal foundation and are in many cases ambiguously described. Hence we are faced with the challenge of interpreting the patterns as well.

This section deals almost exclusively with control flow patterns. Incorporating data patterns, resource patterns and service interaction patterns in our language is future work.

3.4 Workflows as Programs

It would seem that workflows could be expressed in a structured programming language such as Java, and indeed many workflows are implicitly embedded in existing programs. However, it is useful to examine the differences more closely.

Goto revisited Workflows often use *arbitrary cycles*, a pattern akin to goto in structured programming. Goto is traditionally considered bad programming style for two reasons: (1) it makes programs more difficult to understand and debug: if the program may jump to the current location from any other location, very little is known about the state at that point; (2) goto can be eliminated through the combined use of while, if, and function calls.

These two objections are present, albeit less saliently so, for workflows as well. Workflows are most often described using graphical tools meaning that a goto is *explicitly* represented as an arc. Debugging is therefore made somewhat simpler because one immediately can examine the origins of any incoming arcs. Workflow systems should also support equivalents to whiles, ifs, and function calls making it possibly to rewrite the workflow without use of goto. However, for graphical tools it is not always desirable to factor out parts of the workflow in small functions because this may obscure the overall structure of the workflow. This can be alleviated, though, with a language that uses functions instead of goto and displays functions inline or abstracts away from them at the designer's discretion. Such a language should be preferred if possible because it makes formal reasoning more accessible.

Concurrent[6] execution The workflow designer strives to parallelize as much as possible because this affords flexibility: It minimizes the longest path, permitting a runtime trade-off between completion time and resource load at the discretion of workflow manager. It gives flexibility in scheduling because more tasks and sub-workflows can be started without waiting. For the same reason it allows better resource exploitation and less distribution constraints. On the downside, the subtleties of concurrent programming, including deadlocks, livelocks and race conditions, persist.

Although it seems natural to require explicit concurrency constructs in the workflow model, this is not strictly necessary. An implicitly concurrent model simply specifying sequential constraints such as $a\,\texttt{before}\,b$, $all\,[a_1,\ldots,a_n]\,\texttt{before}\,b$ or $one\,of\,[a_1,\ldots,a_n]\,\texttt{before}\,b$ could conceivably express the same. In conventional programming one often employs parallelism for performance reasons starting with a sequential program and splitting it up. In workflow design one approach is to begin with a fully parallel workflow and impose constraints as they become necessary. These observations suggest a programming model where concurrency is the rule rather than the exception.

The presence of concurrency prompts a discussion of shared memory vs. message-passing. This is postponed to future work concerning data patterns.

[6] We take the definition of the Solaris 10 Software Developer Collection Multithreaded Programming Guide [64] namely that: Parallelism is "[a] condition that arises when at least two threads are executing simultaneously" and concurrency is "[a] condition that exists when at least two threads are making progress. A more generalized form of parallelism that can include time-slicing as a form of virtual parallelism."

Delegation Delegation is the allocation of tasks to agents. When many workflows are running, a number of tasks will be available for allocation and a number of agents—human or not—will be ready to take on tasks according to their skill level, workload, availability, speed etc. In the programming language analogy, workflows are communicating multithreaded programs running inside a workflow engine that can call out on several processors to complete the atomic tasks being solicited. Thus, while not directly a part of workflow programming model, delegation affects the architecture of a workflow system. The interplay is best seen in the difference between *explicit* (or *data-dependent*) *choices* made by the workflow engine and *implicit* (or *deferred*) *choices* made implicitly by soliciting both branches for allocation and removing the non-chosen branch, once a task has been started in the other branch. The delegation layer sits between the agents and the running workflows, and essentially works as a scheduler. The solicitation part is the defining difference between delegation and operating system scheduling on multi-processor systems: the tasks may be delegated, solicited or even auctioned off to the processors (agent).

Cancellation Cancellation means aborting a selected number of threads and/or activities and returning to a well-defined state. It has proven particularly tricky to imitate in current workflow systems [70, 67] large because they were based on Petri nets, flow charts, etc. that do not have a natural notion of exceptions. Exceptions in distributed systems in general entail serious considerations as multiple peers attempt to agree on a common perception of the distributed state.

There are some simplifying assumption to be made, however. First, it is relatively straightforward to return all threads inside the workflow engine to a well-defined state. What remains are the tasks in progress outside the workflow engine itself. In a transactional setting these can be assumed to be willing to accept a cancellation at any point with no further obligations. Unfortunately, this will rarely be the case. Cancellation of an almost completed outside task could easily produce an aftermath in the form of e.g. partial invoicing.

As we go from atomic tasks with a simple request/response interface toward tasks with a more extensive interface, ACID[7] issues become apparent. Cancellation is the pattern that shows this most clearly. Also, should we wish to depart from the idea of a centralized workflow management server, we again face all the problems of distributed transactions. Most likely, the solution is a hybrid. Internal workflows can be canceled easily and outsourced parts will have a more elaborate cancellation protocol.

Unconventional control flow patterns Some control flow patterns spawn a number of threads, but only wait for one of a few of them to finish before continuing. The remaining threads may be allowed to finish or they may be canceled.

Skip, redo, and undo tasks In workflows we can randomly decide to skip, redo, or undo tasks. In conventional programs this is inconceivable, because the the data output

[7] Atomicity (either all or no tasks are completed), consistency (data integrity constraints hold before and after transaction), isolation (intermediate states are invisible to the outside world), and durability (a reported success is not reverted).

of that task would be missing. In a workflow system, we can adopt a more lax attitude and simply prompt the user for it when the need arises. This distinction vanishes, of course, when parts of the workflow are automated.

Dataflow for only selected tasks Some dataflow patterns break with a venerable tradition, and propose that data flow be completely orthogonal to the control flow, e.g. a task can decide to forward its output to another task—and *only* this other task—several steps down the line. So far no workflow languages have adopted such a liberal data flow model, and the advantages relative to the increase in complexity are not obvious. Certain security scenarios can be neatly expressed with directed data flow, but these can also be solved with access control lists or a role scheme.

3.5 Control Patterns

This section is based in part on Christian Stefansen: *SMAWL - A SMAll Workflow Language Based on CCS* [62, 63].

3.5.1 Why Consider a Process Calculus

In 2003-2004 there was an intense debate [81, 73][8] in the workflow community about the relative merits of Petri nets [51] and π-Calculus [44] for workflow modeling. Nonetheless, very little of this debate has been published or put in writing elsewhere. During the discussions, van der Aalst presented seven challenges [73] to model workflow in π-Calculus. This section (based on a previous paper [62, 63]) 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. The applicability of π-Calculus to the workflow modeling domain is briefly discussed and a new *overlay* operator is discussed with applications to workflow descriptions. The central challenge is this:

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

Whatever one's beliefs, a concrete encoding of the 20 workflow patterns provides a basis for a more informed debate and makes it easier for implementers to make a choice of formal foundation. The choice of the π-Calculus here is more or less arbitrary. A vast number of other calculi (e.g. the Join calculus) would be equally interesting.

The π-calculus and its predecessor CCS (Calculus of Communicating Systems) [43] have strong mathematical foundations and have been widely studied for years. This has lead to a plethora of deep theoretical results (see [58] for an overview) as well as applications in programming languages (e.g. PICT [52]), protocol verification, software model checking (e.g. Zing [6, 80]), hardware design languages[9], and several other areas.

[8] Only two references may not support the claim of an "intense debate" very well. There were many more papers, most of which were so speculative and agenda-driven that we will not dignify them by reference.

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

Workflow description languages, it would seem, have a lot in common with these areas. First, workflows can be thought of as parallel processes. Second, workflows often defy the block structure found in conventional programming. Third, 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. In particular providing understandable graphical tools and user-friendly abstractions constitutes a pivotal challenge if π-calculus-based systems are to succeed.

In the same vein a fair concern to raise is whether the level of formalism in π-calculus is necessary considering how graphical systems today are already difficult to understand both for users and programmers. First, using π-calculus does not preclude a high-level graphical representation, and clearly more research into this area is warranted. Second, a rigorous foundation remains necessary, because we strive to understand systems *before* we implement them. A well-tested, well-understood foundation will make implementation and subsequent adaptation easier and more routine.

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. It may turn out, though, that channel-passing (and possibly locality) will become highly relevant when addressing service infrastructure and actual implementation of workflow tasks. By choosing π-calculus we stand a better chance of finding a unified foundation for business systems programming.

In terms of practical workflow management π-calculus promises seamless integration between workflow systems and existing verification tools (model checking [11, 29], behavior types/conformance checking [54, 36] etc.). For the end-user this means access to verification tools that will make writing and adapting workflows faster and less error-prone.

In our view it is important that future business systems are built not only to carry out a specific task, but in a way that makes them amenable to automatic source code manipulation and formal verification, and to attain this goal it is desirable to have a small and rigorously defined core model. What we are seeking is not a formalism to use as it is, but the formalism that will provide the most suitable underpinning for future work.

For the purpose of workflow control patterns we can afford ourselves a limited view of the concurrency models available. Many process calculi, including π, Join, CCS, and Ambients whether synchronous or asynchronous retain simplicity by assuming global consensus, i.e. the existence of a global mechanism ensuring that only one process consumes a message sent on a channel with multiple listeners. It is useful to start

out with a centralized workflow management system and deal with distribution issues separately later.

The following patterns may look a bit daunting when first presented, but one should remember that they are not what the workflow user/programmer should see any more than object calculus terms are what a Java programmer or UML user sees.

3.5.2 Modeling Control Flow in CCS

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

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

$\mathbf{0}$ is the empty process, τ is some unobservable transition, $a!x$ and $a?$ try to send and receive on channel a, $+$ is choice, $\mid$ 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[10]. More sophisticated protocols for monitoring activity progress can be added, but here 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.

It is important to notice that the workflow patterns coded here are abstract patterns that do not make any assumptions about the data flow. This separation of concerns, besides being an important design principle, allows a clear and focused comparison without too much clutter. It also means that one can plug in one's data-flow language of choice at any later point. The analysis made here holds *regardless* of the data-flow language chosen, not just for one specific data-flow language, and so in effect the analysis is stronger and more general because of this abstraction[11].

3.5.3 The Control Flow 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 [69] (see Table 3.1), and the pattern descriptions are taken verbatim from [68] with minor clarifications in square brackets.

[10] The mechanism does not work if multiple instances of a are invoked simultaneously. We ignore this for the moment.

[11] Afterwards, a similar study could be made with regard to data-flow requirements, and of course any realistic system should handle both control flow and data flow.

Table 3.1 The 20 Workflow Patterns [68] and Two New Ones

Basic Control Patterns	**Patterns Involving Multiple Instances**
1 Sequence	12 MI without synchronization
2 Parallel Split	13 MI with a priori known design time knowledge
3 Synchronization	14 MI with a priori known runtime knowledge
4 Exclusive Choice	15 MI with no a priori runtime knowledge
5 Simple Merge	
Advanced Branching and Synchronization Patterns	**Structural Patterns** / **Cancellation Patterns**
6 Multiple Choice	10 Arbitrary Cycles / 19 Cancel Activity
7 Synchronizing Merge	11 Implicit Termination / 20 Cancel Case
8 Multiple Merge	**State-Based Patterns**
8a N-out-of-M Merge ***(new)***	16 Deferred Choice
9 Discriminator	16a Deferred Multiple Choice ***(new)***
9a N-out-of-M Join	17 Interleaved Parallel Routing
	18 Milestone

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 does 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 [43]: 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. □

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

$$P_1 \mid \cdots \mid P_n$$ □

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

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

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. □

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 $+$. □

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

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

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. □

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. □

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

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

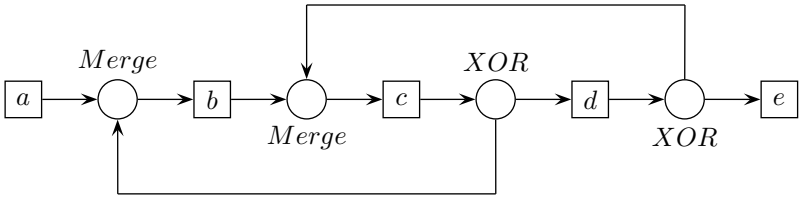
Here the *Multiple Choice* pattern is used but other split patterns could be used too. The first signal on ok initiates Q and all remaining signal are consumed. Contrary to Petri nets, ignoring all future signals is not a problem because any loop around this construct would simply instantiate a new process (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 [68]:



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

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

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 $\mathbf{0}$. 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 [43]), (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 will 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:

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

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:

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

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). □

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 $isoff?$:

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

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 [68] 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. □

3.5.3.1 More Split and Join Patterns

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 3.2 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. If equipped with simple constructs for writing such predicates, the language becomes much more expressive than the patterns here, but for analysis purposes it is very important that the predicates remain within the boundaries of Presburger arithmetic or some other (preferably simpler) first-order theory in which all statements are decidable. This idea is examined in detail below.

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. □

Table 3.2 Split, Synchronize, and Merge Patterns. $l \rightarrow n/m$ means "control flows from l path(s) to n out of m possible paths". $n/m \rightarrow l$ means "control flows from n of m possible paths into l.

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

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):

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

□

3.5.4 Simplifying the Patterns

Ideally, the control flow patterns should have a formal representation. Such a representation should ensure that the patterns have a clear semantics, are atomic, have a minimum encoding bias (i.e. with respect to any particular concurrency model), have minimum overlap, and are as simple as possible. The current description format was clearly devised with a different set of goals in mind. Several overlapping patterns were added for easy of benchmarking. E.g. the presence of four multiple instance patterns instead of just one allows a finer benchmarking of systems that do not support the most general pattern. Also, the patterns were developed in response to the previous very limited research asserting that split, join, loop, and sequence were sufficiently expressive for all workflows. Thus, an (industry) accessible description format was needed.

Formal descriptions are important, however, as they ameliorate further research: In the current state of affairs no language can reasonably make the claim of supporting all patterns. Although we can make this statement plausible by meticulously encod-

ing every pattern in isolation as we have done, there is no theorem showing that every conceivable combination of the patterns can also be expressed. Formal descriptions also open the doors to verification, simulation, rigorous comparison, compositional analysis, and algebraic reasoning. First and foremost, a formal description of the patterns is an important step towards designing a programming language for workflows. Currently, the only formal description is the one implicit in the implementation of the YAWL system [67] but this does not satisfy the minimum coding bias criterion—neither do Petri net-inspired graphical illustrations. Before considering a formal representation a brief analysis of the patterns is in place.

In their current state the patterns do not live up to the criteria sketched above. A closer look at the patterns, reveals that they straddle graphical and algebraic approaches, and that attaching a semantics is non-trivial. Some patterns do not seem atomic. The "Multiple instances without a priori runtime knowledge" pattern contains both iteration, splitting, and merging in one pattern. On the other hand the "Synchronizing Merge" pattern has no well-defined semantics in isolation simply because it is not known how many completion signals it should wait for. Non-trivial overlap also exists: "Multiple Merge" could be expressed as "Sequence" without explicit synchronization, and if instead "Sequence" is taken to imply tacit synchronization then it would seem that a pattern for explicit synchronization was no longer needed.

Let us return to the idea of workflows as initially completely unstructured. Tasks $a_1, \ldots, a_n$need to be performed, and every workflow pattern is a different way of imposing constraints (w.r.t. ordering, data-dependency, time, etc.) on the task list. The question is how to express these constraints. Curbera et al. [12] identify two major directions: the graph-oriented approaches and the algebraic approaches. Examples of the former include Petri net-based models (most significantly YAWL [67]), Event-Driven Process Chains (EPCs) [59], and dependency graph models (e.g. GNU `make`[12]). Examples of the latter include—most prominently—models based on π-Calculus and Join Calculus. According to Curbera et al. BPEL [65]—arguably today's most widespread standard—is a hybrid approach, which combines ideas from WSFL (graph-based, loosely based on Petri nets) and XLang (algebraic, loosely based on π-Calculus). This distinction may reveal the starting point of the language designers and their preferred type of formal reasoning, but what is more interesting are the restrictions on the language relative to the patterns. As we have seen, all the control flow patterns can be encoded in a process calculus as they are. Curbera et al., however, seem to suggest that the algebraic approach enforces a strictly nested syntactical regime, e.g. where all splits and join match one-to-one by design of the constructs. The algebraic approach does enforce a syntax tree, naturally, but this does not imply any of the constraints suggested on the control flow graph unless chosen to by design. A more rigorous taxonomy of workflow models is needed.

Let us frame this discussion by considering some challenging examples. All these examples can be given a strict semantics if disambiguating hints are provided by the workflow designer, but not all models will be able to express them (i.e. without heavy reliance on some mechanism for shared state). The goal, however, is a minimal burden on the workflow designer (hopefully yielding the corollary "minimal opportunity to

[12] `http://www.gnu.org`

Figure 3.1 Van der Aalst's *Challenge 4* [73]

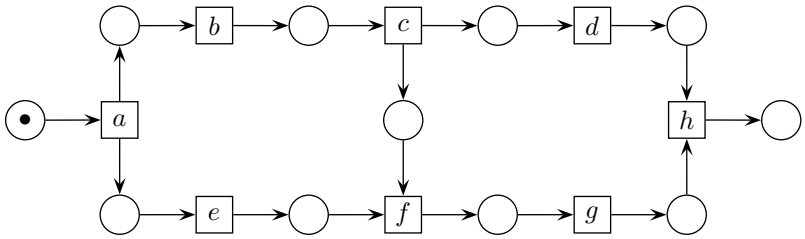
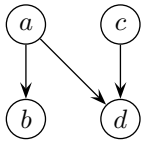


Figure 3.2 Non-nested control flow



introduce bugs"[13]).

3.5.4.1 Non-Matched Splits and Joins (No Block Structure)

As mentioned previously a major difference between workflows and the control flows found in conventional programs is the widespread lack of block structure. Consider the Petri net in Figure 3.1. The Petri net exhibits a pattern that is cumbersome to express in workflow languages that only have block constructs. The core problem is the pattern shown in Figure 3.2. On the other hand, graphical models such as Petri nets or even GNU `make` have no problems expressing this. The example was provided as a challenge to the π-calculus community during a 2004 surge in interest for process algebraic approaches to workflow modeling. XLang, for instance, claims to be based on the π-calculus and indeed has constructs for sequence, choice, parallel flow, and iteration (`sequence`, `switch`/`pick`, `all`, and `while`, respectively). It adopts an implicit synchronization scheme for the sequence operator similar to the idea developed in our encoding of the *Sequence* pattern. XLang lacks internal message primitives, however, and this makes the link between c and f impossible to express[14]. The conclusion is: a language with block structure only, will not be sufficient. There are several remedies available, of which internal message-passing immediately springs to mind. First, though, let us consider another idea. We previously [63] conjectured the usefulness of an *overlay operator* inspired by the parallel operator $||$ found in Communicating Sequential Processes (CSP) [27]. The overlay operator $\&$ is defined by the rules in Figure 3.3. The major difference between $\&$ and $||$ is the "no mention" rule (the first rule), which allows P to transition on α if Q does not men-

[13] In an abuse of Alfred North Whitehead's famous quote we could say that the ultimate goal of computer science is to eliminate to need for intelligent thought.

[14] The link between c and f could of course be simulated through use of shared data. The statement of impossibility made here holds only in the absence of controlled access to shared data. A formal (Petri net-based) proof of such an expressiveness result can be found in Kiepuszewski's Ph.D. dissertation, theorem 5.3.2, pp. 139-141 [34].

tion α. Thus the $||$ operator can be obtained by removing the first rule. The reduction semantics may seem convoluted, but it is worth noticing the neat trace semantics: $traces(P\&Q) = \{t : t_{\alpha P} \in traces(P) \wedge t_{\alpha Q} \in traces(Q)\}$, where $t_{\alpha P}$ denotes the projection of the trace t onto the ports of P[15]. The operator is perhaps best explained by coding the example in Figure 3.1:

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

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 does not impose constraints on that transition. Here if a, e was 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 interesting, not only as a remedy for block-structured languages, but also because it allows the overlaying of global business rules to all processes. E.g. one might have a rule that says "delivery shall occur before invoicing". This rule would now simply be enforced on all workflows by overlaying them with the rule workflow. Also, the workflow designers do not have to describe all business rules repeatedly because the can simply overlay the global rules to their workflows. Of course, this requires a strict nomenclature of tasks in order to work, and possibly something more flexible than simple alphabetic matching would be needed.

BPEL has a different solution for the problem. As mentioned BPEL combines the block constructs of XLang with the free graph ideas of WSFL. In addition to block constructs, BPEL allows dependencies in the form of named *links* between nodes, each of which should acknowledge participation in the link by including a `source` or `target` tag containing the link identifier. In BPEL, challenge 4 would most likely be solved by creating the link from c to f using its a `source` tag in c and a `target` tag in f. Links are limited, however, as they may not extrude or intrude while loops, they may not introduce cycles, and they are governed by several other syntactical restrictions. BPEL therefore does not offer first-class support for arbitrary cycles.

Compared to BPEL's notions of links through `source` and `target`, the $\&$ operator offers separation of concerns: the source and target nodes do not explicitly express the link because it is factored out. Whether non-local logical constraints in the style of the $\&$ operator is a good or a bad idea, remains an open question pending practical studies.

It should be stressed, that the operator remains speculative at this point and has not been developed formally in a CCS setting. The operator can be built into the reduction rules of a workflow system, or it can be translated down to regular CCS. It is not important whether the introduction of $\&$ adds expressiveness[16] to CCS itself, but an $\&$-like construct can strictly improve expressiveness for more limited high-level workflow languages.

[15] The trace semantics of $||$ is given as $traces(P||Q) = traces(P) \cap traces(Q)$.

[16] In Felleisen's definition of expressiveness [17].

Figure 3.3 Reduction Rules for the Overlaying Operator &

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

3.5.4.2 Multiple Merge and Multiple Instances

Another dividing concern is that of multiple instances. There are two sources of multiple instances: (1) using the *Multiple Merge* pattern to join paths from a previous split pattern and (2) through the multiple instance patterns 12-15. Kiepuszewski argues that using *Multiple Merge* is a convenient and natural way of expressing multiple instances [34]. Whereas this may hold for graphical languages, it is not necessarily desirable for algebraic languages. For instance it is far from obvious that an idiom such as

$$merge(parsplit(A, B, C, D), E)$$

is really needed. Disallowing *Multiple Merge* and instead appending a function call to each branch would not seem like an unreasonable burden, since there is a fixed and typically low number of branches.

In case (2) where we explicitly spawn multiple instances, there is clear need. First, because expressions such as $split(P, P, P, P)$ are tedious even if P only consists of a function call; second, because there could easily be a high number of Ps; and third, because the final number of branches is not known design-time or even runtime (cf. pattern 15: *Multiple Instances With No A Priori Runtime Knowledge*). With the exception of the runtime generativity behavior (i.e. patterns 14 and 15), the regular split/join patterns would seem able to capture multiple instance patterns with some few alterations. We will explore this in more depth in section 3.5.4.4.

BPEL supports the multiple instance patterns only through scant workarounds [1], and BPEL also does not support *Multiple Merge*.

On a different note, there is an overlap between *Sequence*, *Multiple Merge* and synchronization. Consider the expressions

$$merge(parsplit(A, B, C, D), E) \tag{3.1}$$

$$sequence(parsplit(A, B, C, D), E) \tag{3.2}$$

$$synch(parsplit(A, B, C, D), E) \tag{3.3}$$

There are two interpretations: (1) All four threads A, B, C, D should spawn their own copy of E or (2) when all threads A, B, C, D are done, one instance of E should run. However, these two interpretation are split across three control flow patterns, and hence one is unnecessary. Either we assume that $sequence$ in expression 3.2 implicitly joins the four threads (interpretation 2). In this case the $synch$ pattern is unneeded. Alternatively, we assume that $sequence$ in expression 3.2 does not join the four threads (interpretation 1). In this case the $merge$ pattern is unnecessary. The question is this: should synchronization be explicit or should merge be explicit? Merge is by far the most dangerous construct to forget, thus strongly suggesting that merge should not happen implicitly. The existence of such overlaps underlines the need for a formal representation of the patterns.

Figure 3.4 The Travel Agency: *Synchronizing Merge* in action

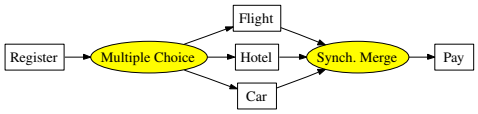
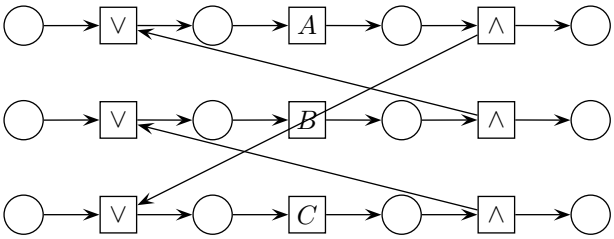


Figure 3.5 OR-join nightmare scenario (example from Figure 3, p. 12 [35])



3.5.4.3 Synchronizing Merge

As it turns out, a lot of the problems with the patterns can be framed neatly by looking at the "Synchronizing Merge" pattern.

Synchronizing Merge is described as "merge many execution paths. Synchronize if many paths are taken. Simple merge if only one execution path is taken." Before continuing consider Figure 3.4 (The Travel Agency) for a motivating example[17]. Synchronizing Merge occurs far too often to be ignored, and indeed there is nothing contrived about the the Travel Agency example. At runtime a user (data-dependent, explicit) choice is made as to whether flight, hotel, and/or car reservations are needed. Once the reservations go through, the task *Pay* is activated, and, of course, the join should only wait for the branches that were actually activated.

The problem is that Synchronizing Merge has non-local semantics; it cannot be given a semantics without either

1. integrating it with the preceding split patterns
2. leaving the problem to the runtime system or
3. establishing a method of propagating information about the paths actually taken in prior splits to the Synchronizing Merge.

This well-known problem appears in the context of high-level Petri nets [69], Event-Driven Process Chains [74, 35, 59, 55], and BPEL4WS [12]. The problem is more complicated in the presence of loops because the semantics of a Synchronizing Merge can be dependent on a split further down the line. The problem can be alleviated in a variety of ways. Runtime checks, syntactic constraints, message-passing etc. may work, but as we have seen, even a seemingly unobtrusive idea such as imposing a block structure has a serious impact on the expressiveness.

Event-Driven Process Chains [59] are probably they least constrained model: EPCs define the concepts of *functions* (called tasks is this report), *information objects* (data), and *events* (events or signals) in a graphical language where functions and events constitute the nodes. Functions and events can be connected using arcs; an arc from an

[17] This example was initially suggested by Wil M.P. van der Aalst.

event to a function stipulates that the event be triggered before the function is executed; an arc from a function to an event means the event is fired upon completion of the function. In addition to the function and event nodes there are logical connectives $\wedge$, $\vee$, and XOR. With only a few restrictions, arcs can connect events to connectives, connectives to events, functions to connectives, and connectives to functions.

EPCs form the basis of SAP R/3's workflow modules and the ARIS (Architektur integrierter Informationssysteme) framework. EPCs have great modeling power, thanks to the unconstrained graphical form (allowing arbitrary cycles) and the $\vee$-connective, which expresses the idea of the Synchronizing Merge pattern. EPCs were initially proposed in an informal fashion, and a long discussion about the precise semantics of the $\vee$-connective ensued. Finally, it was made clear rigorously, that while EPCs can informally express all the examples of this section and more, EPCs have serious problems with formal semantics [74]. Indeed, an attempt to express the semantics as fixed points of the transition relation showed that neither greatest nor least fixed points are necessarily unique or exist [35]—clearly this is not the way to go.

YAWL relies on the runtime system for the semantics of Synchronizing Merge: The OR-join transition is allowed "if the number of consumed tokens cannot be increased by postponing the occurrence of the OR-join" [75]. The formal definition can be found in the same report p. 30. YAWL is designed to run on a centralized server that stores the entire workflow state and therefore can make judgments about the OR-join. In a distributed setting, the workflow resides on different servers, and deciding whether an OR-join is allowed or not may induce too much overhead – the non-local semantics again becomes a problem. Consider Figure 3.5. Assume that each horizontal thread of execution resides on a different server, and observe the cyclic dependency. One semantics could be to solicit all three OR-joins for execution. Once one join is taken, the others should no longer be possible, because there is a way of eventually feeding both incoming arcs on them. The creates a global consensus problem, that is unacceptable in distributed settings unless kept at a strictly minimal level of occurrence.

BPEL also faced the Synchronizing Merge problem, but adopted another approach. In BPEL information about non-chosen paths is propagated by *dead path elimination* (DPE). A join node initially has the value *unevaluated* assigned to each incoming link. If at runtime it becomes clear that the link cannot be activated, the value *false* is assigned to it. This is done by propagating *false* down through a branch once it is clear that another alternative was chosen[18]. If a link can be activated, eventually it will be or the link will become tagged as *false.* The join condition is satisfied once all incoming links are evaluated to either *true* or *false*. Essentially, this means that in BPEL Synchronizing Merge has been replaced by a runtime facility. A signal is always received, and its boolean value decides the semantics. Unfortunately, BPEL imposes heavy restrictions on the propagation of (internal) signals and therefore it cannot be said to be as general as YAWL.

The *Synchronizing Merge* pattern is difficult to avoid without serious setbacks in expressiveness. At the same time, any system allowing it must employ some variation of the three solutions suggested (add structure to workflow, propagate information or

[18]Provided the flag `suppressJoinFailure` is set – otherwise an exception is raised explaining that the process has become stuck.

decide runtime). As we have already argued, requiring block structure—even locally—is unacceptably restrictive. Propagating information only works in limited settings, and breaks down when cycles are present. This is clearly seen by the number of constraints on links in BPEL. A runtime rule such as the one defined for YAWL seems to be the proper solution for single-server systems, but perhaps a less restrictive propagation semantics could be designed for future versions of BPEL. The last word about *Synchronizing Merge* is yet to be said.

3.5.4.4 Generalizing Split and Join Patterns

Synchronizing Merge is the *only* one of the split and join patterns that does not have a strictly local semantics. This observation inspires two ideas with regard to the remaining split and join patterns:

1. Split and join patterns could be expressed more generally with a grammar for what we could call "split and join conditions".
2. Multiple Instance patterns could be split up and merged into the general way of expressing split and join conditions.

It should be noted though, that some of the advanced Multiple Instance patterns (*with a priori runtime knowledge* and *without a priori runtime knowledge*) in some cases require a *Synchronizing Merge*.

The split patterns (*Exclusive Choice*, *Deferred Choice, Parallel Split*, *Multiple Choice*) can be covered in a number of ways. E.g. it would be enough to simply have a generic

$$split(min, max, [P_1, \ldots, P_n])$$

where min and max denoted the minimum and maximum number of branches that needed to be started. Thus *Choice* would have $(min, max) = (1, 1)$, *Parallel Split* would have $(min, max) = (n, n)$, and *Multiple Choice* would have $(min, max) = (1, n)$. The distinction between explicit and implicit (deferred) choice would be in guarding the P_i with predicates. This pattern is far more general that any of the patterns it tries to encompass. First it allows mixing guarded and unguarded processes, i.e. mixing explicit and implicit choice. Second, (min, max) can take on different values to signify limits branching limits. So far we have tacitly assumed that every P_i would be run at most once, but this does not have to be the case. An even more generic form could take this into account with a minimum and maximum for each branch:

$$split(min, max, [(P_1, min_1, max_1), \ldots, (P_n, min_n, max_n)])$$

Thus Multiple Instances can be expressed as $split(1, \infty, [(P, 1, \infty)])$, and a host of other variations are possible (along with a wide range of new sources of errors). Once again this covers significantly more territory than the patterns it was intended to model.

When is the split completed? If $min \neq max$, the semantics need further specification. Even more so if some P_i are predicate guarded and some are not. Every P_i can invoke another P_j during its execution, but at some point the creation of threads should be considered completed, even if the maximum has not been reached. In other words, if a split has spawned between min and max branches, it should solicit tasks for execution, until there is no longer any need. This need is decided by the synchronization

side. Let us therefore consider the synchronization patterns. Here we can equivalently conjure up a syntax such as

$$synch(waitfor, P)$$

to denote that all active branches of P should be synchronized, but control should be passed on, once $waitfor$ branches have completed. Thus assuming P has n branches, we can write $synch(n, P)$ (*Synchronization*), $synch(1, P)$ (*Discriminator*), or $synch(m, P)$, where $1 < m < n$ (*M-out-of-N Join*). Following the observation in section 3.5.4.2, it is probably safer to require explicit merge rather than explicit synchronization. Hence $synch$ should become an implicit part of $split$ and in addition there should be a way of doing a merge.

What was just presented is a highly speculative design, and only one of many possible designs. The min and max designations are merely a simplified predicate language describing the allowable splits from a universe of options. An equally workable solution, therefore, is to simply introduce a minimal predicate language, very much like connective nodes of EPCs or the join conditions of BPEL. This especially makes sense for joins, where we need to answer two questions: (1) when can control be passed on, and (2) when should all remaining processes be canceled and/or garbage collected. For splits we usually need to answer the question: when it the split done spawning branches. As long as branches can be spawned, the join side cannot do a *Synchronizing Merge.*

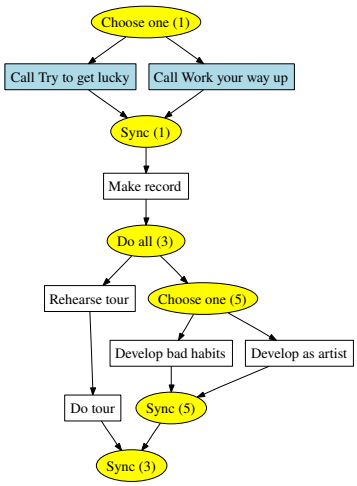
Where do we stop? This should not degenerate into a race for more patterns, in turn demanding more expressiveness of an already overburdened set of constructs. The language must be kept simple, and already at this point, the convenience of a simple reduction semantics seems far away. Conversely, the language needs to present a few comprehensible constructs rather than an array of partially overlapping patterns. Thus, the hunt for a simple language is certainly justified. Pinpointing the exact design criteria for a workflow language is future work. The existing patterns are justified by observation in real-life scenarios, and, as we have seen, there are many ways of designing languages to express them all. What is not acceptable is the current state of the patterns, where part of the generativity is camouflaged in the *Multiple Merge* pattern and some is packaged in the protective padding of the *Multiple Instance* patterns. YAWL for instance claims to support all patterns, but YAWL would not be able to handle a *Multiple Instance* pattern with a *Discriminator* join, since all the *Multiple Instance* patterns described currently, use either Multiple Merge or Synchronization to join. Therefore the line of research to clean up the patterns and furnish them with a formal semantics should continue. This effort should be two-pronged, simplifying high-level descriptions on one side while developing a suitable low-level calculus on the other side.

3.5.4.5 Other Patterns

The *Arbitrary Loop* pattern requires a non-block structure as well as repetition. We can solve this using BPEL-style links. Alternatively, it would be enough to allow functions (essentially merge points), synchronization, and forks without synchronization. The idea of functions or sub-workflows seems oddly missing from the patterns. The merge

Figure 3.6 How To Become a Recording Star (adapted from the *Recording Star* example *[68])*

```
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
```



patterns buy us a lot in terms of code reuse, but it does make the patterns seem heavily slanted towards graphical languages (read: Petri nets).

Explicit state in workflow systems is essential to differentiate between deferred (implicit) and active (explicit) choice. Petri nets represent state as places, and the π-calculus represents state as process expressions.

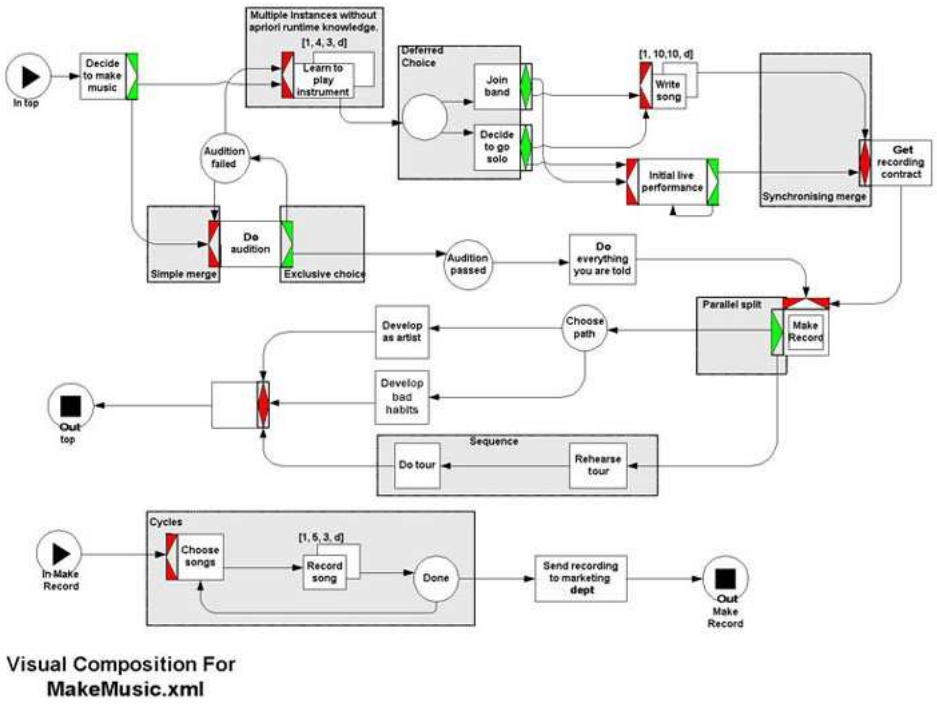
The cancellation patterns are the biggest joker. YAWL has an operator that clears all tokens from a subnet, BPEL has exceptions, LOTOS [8] has the disrupt operator, and none of these have made a convincing argument, that they will work in a distributed setting. Neither Petri nets nor π-calculus handle cancellation patterns very nicely. Cancellation patterns are ubiquitous. Not just because of failure in distributed workflow, but because the *Discriminator* and the *n/m-Merge/Synchronization* patterns conceivably invoke cancellation methods on the remaining threads to ensure graceful termination.

3.5.4.6 Towards a Workflow Description Language Based on CCS

The workflow patterns coded in the previous sections show that CCS is too low-level to be used directly as a workflow language, and that certain patterns are very cumbersome to write. The language needs high-level constructs and a more pleasing syntax. The goals should be (a) to reduce the amount of user-specified internal synchronization mechanisms and (b) to provide elegant constructs for the 20 (+2) workflow patterns. The language SMAWL, presented here and in previous work [62, 63], seeks to do exactly this while maintaining a strong link to CCS. The example in Figure 3.6 based on [68] shows what a workflow specification might look like in SMAWL along with an automatically generated[19] graphical representation.

[19]Using a Standard ML program to map the abstract syntax tree to a `dot` file, and running `dot`.

Figure 3.7 Recording Star example in YAWL (taken from [68])



Having completed the analysis above, a natural idea is to design a language out of small building blocks of based on some of the patterns. This way there would be, e.g., a number of split combinators and a number of join combinators. However, as mentioned in section 3.5.4.2 explicit synchronization every time a split block is left is tedious and possibly dangerous; a more pleasing style is therefore to implicitly synchronize after every split construct and have the programmer explicitly write if a continuation should be spawned for each active thread (a merge). These and numerous other considerations lead to the following syntax:

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

In the syntax ϵ denotes the empty string, f is a function name, w is the workflow name, ρ is a data-dependent predicate[20], 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.

[20]The format of predicates ρ is not of the essence here; such predicates will simply be compiled to a τ

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 expand our example. The example shown in Figure 3.8 is based loosely on the Petri net example at [68] to make it easier to compare the Petri net approach to a CCS approach.

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 $\mathcal{T}[\![\cdot]\!]$ (see figure 3.9) 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.

The transformation of the **cancel** construct makes use of the function $\mathcal{C}[\![\cdot]\!]$, which inserts 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:

prefix and the responsibility of deciding them will be delegated to a data-aware layer. The idea of parameterizing over the predicate/expression language is also found in BPEL, which can use XPath or another plugged-in language.

Figure 3.8 A Larger Example: How To Become a Recording Star

```
workflow Becomearecordingstar =
    chooseone {
        ⇒ call (Workyourwayup)
        ⇒ call (Trytogetlucky)
    };
    Makerecord;
    doall {
        ⇒ chooseone {
            ⇒ Developasanartist
            ⇒ Developbadhabits
            }
        ⇒ Rehearsetour;
            Dotour
    }
end

fun Workyourwayup =
    multi {Learntoplay};
    chooseone {
        ⇒ Joinaband
        ⇒ Decidetogosolo
    };
    chooseany {
        ⇒ multi {Writesong}
        ⇒ multi {Performlive}
    }
end

fun Trytogetlucky =
    Doaudition;
    chooseone {
        ⇒ Audition.failed
            chooseone {
                ⇒ call (Trytogetlucky)
                ⇒ call (Workyourwayup)
            }
        ⇒ Audition.passed
            Doeverythingyouaretold
    }
end
```

Figure 3.9 The Transformation $\mathcal{T}[\![\cdot]\!]$ from SMAWL to CCS

$$\mathcal{T}[\![\textbf{workflow } w = P \textbf{ end}]\!] = \mathcal{T}[\![P]\!]$$

$$\mathcal{T}[\![\textbf{fun } f = P \textbf{ end } D]\!] = \lambda ok.f?*.\mathcal{T}[\![P]\!]f \mid \mathcal{T}[\![D]\!]ok$$

$$\mathcal{T}[\![\textbf{milestone}(ison, isoff, set, clear)\ D]\!] =$$
$$\lambda ok.\text{Milestone}(ison, isoff, set, clear) \mid \mathcal{T}[\![D]\!]ok$$

$$\mathcal{T}[\![\textbf{newlock}(l, u)]\!] = \lambda ok.\text{let } unlocked \Leftarrow \nu() \text{ in let } locked \Leftarrow \nu() \text{ in}$$
$$unlocked? * .l?.locked! \mid locked? * .u?.unlocked! \mid unlocked!$$

$$\mathcal{T}[\![\textbf{lock}(l, u)\{P\}]\!] = \lambda ok.\text{let } ok' \Leftarrow \nu() \text{ in } l!.\mathcal{T}[\![P]\!]ok' \mid ok'?.u!.ok!$$

$$\mathcal{T}[\![activity]\!] = \lambda ok.activity!.activity?.ok!$$

$$\mathcal{T}[\![\textbf{send } f]\!] = \lambda ok.f!.ok!$$

$$\mathcal{T}[\![\textbf{receive } f]\!] = \lambda ok.f?.ok!$$

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

$$\mathcal{T}[\![\rho\ P]\!] = \lambda ok.\tau.\mathcal{T}[\![P]\!]ok$$

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

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

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

$$\mathcal{T}[\![\textbf{call } f]\!] = \lambda ok.f!.f?.ok!$$

$$\mathcal{T}[\![\textbf{cancel } \{P\}]\!] = \lambda ok.\mathcal{C}[\![\mathcal{T}[\![P]\!]ok]\!]$$

$$\mathcal{T}[\![\textbf{multi}(n)\ \{P\}]\!] = \lambda ok.\text{let } create \Leftarrow \nu() \text{ in let } ok' \Leftarrow \nu() \text{ in}$$
$$\underbrace{create!.\cdots.create!}_{n}.\underbrace{ok'?.\cdots.ok'?}_{n}.ok! \mid create?*.\mathcal{T}[\![P]\!]ok'$$

$$\mathcal{T}[\![\textbf{multi}(\infty)\ \{P\}]\!] = \lambda ok.\text{let } inc \Leftarrow \nu() \text{ in let } dec \Leftarrow \nu() \text{ in let } zero \Leftarrow \nu() \text{ in}$$
$$\text{let } loop \Leftarrow \nu() \text{ in let } ok' \Leftarrow \nu() \text{ in let } create \Leftarrow \nu() \text{ in}$$
$$loop! \mid loop?*.(create!.loop! + zero?.ok! + ok'?.dec?.loop!)$$
$$\mid create?*.inc!.\mathcal{T}[\![P]\!]ok' \mid Counter(inc, dec, zero)$$

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

where α 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.

3.5.4.7 Classification of Workflow Models

Given the control flow patterns on one side and a wide swath of potential models on the other side, it behooves us to bring these together on a more formal footing. At least one such attempt has been made:

> "Specifically only two products Verve and Forte Conductor support Pattern 8 (Multimerge) and they have very similar support for the rest of the patterns. We will classify them into the class of Standard Workflow Models with the main characteristic being the ability to create multiple concurrent instances of one activity. In contrast Staffware Changengine and iFlow do not have support for Pattern 8 (MultiMerge) otherwise they are very similar to Standard Workflows. We will classify them as Safe Workflow Models. Both SAP R/3 Workflow and FileNet Visual WorkFlow do not support Pattern 10 (Arbitrary Cycles) and through the Test Harness we have learned that due to syntactical restrictions certain processes cannot be modelled. We will consider these products to be members of a class called Structured Workflow Models. Finally only one product MQ Series Workflow supports Pattern 7 (Synchronizing Merge) and we will consider it a member of a class which we will refer to as Synchronizing Workflow Models." p. 96 [34]

YAWL belongs to the Standard Workflow Models. It does allow *Synchronizing Merge*, but the important observation is that the semantics of the merge is based on global runtime reachability analysis, not on propagation of boolean synchronization signals. The rationale for the name "Synchronizing Workflow Models" for models supporting *Synchronizing Merge* is that they *always* synchronize on all inbound links. E.g. BPEL waits until either a true or false signal has been received on all links before continuing. In essence, the *Synchronizing Merge* problem has been avoided by recasting it as pure *Synchronization*, hence the name Synchronizing Workflow Models. Kiepuszewski proves the interesting result that Synchronizing Workflow Models cannot deadlock! (This result also holds true for WS-BPEL [49]). YAWL, by contrast, does not wait

for a signal from all links; rather its runtime reachability algorithm [45] determines whether a signal will ever arrive or not.

Kiepuszewski [34] argues that Standard Workflow Models should be preferred, because the convenience of generativity through *Multiple Merge* and arbitrary loops supersedes the benefits of ease of synchronization. It is true within Kiepuszewski's framework that arbitrary loops are a serious problem for Synchronizing Workflow Models. However, SMAWL, by virtue of its ties to CCS, offers the options to spawn both synchronized (**call**) and non-synchronized (**send**) threads at any point as well as ad hoc synchronization (**receive**)—while keeping a block structure convention, where all subexpressions deliver explicit termination signals. SMAWL has at least the expressiveness of Standard Workflow Models, but at times it will be necessary to split complex non-nested cycles into smaller functions/sub-workflows

3.5.5 Related Work

Dong and Shensheng presented their encoding of some of the patterns in an unpublished paper in 2004 [16]. 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.

In a paper to appear [53] Puhlmann and Weske demonstrate control flow encodings in π-Calculus using an ECA (Event/Condition/Action) approach. They too routinely invoke channel-passing, without supplying a particular reason. The *Cancel Case* pattern is avoided altogether, and the *Synchronizing Merge* semantics is left to an unspecified runtime component.

3.5.6 Conclusions and Future Work

In this section we

1. discussed the differences between workflows and conventional programs,
2. showed a CCS encoding of the 20 control patterns,
3. presented two new patterns *(16a) Deferred Multiple Choice* and *(8a) N-out-of-M Merge*,
4. hypothesized the value of an overlaying operator,
5. analyzed the 20 control flow patterns,
6. sketched SMAWL, a set of high-level, CCS-based constructs covering all 20 patterns, and
7. commented on the Kiepuszewski's workflow language taxonomy.

All these items aim to widen our understanding of workflow language design. A workflow language should be have minimal encoding bias, be as simple as possible, cover all 20 patterns (and more), lend itself to program analysis and transformation, and have a clear semantics. If we wish to cover all 20 patterns and have a clearly defined semantics, several issues need to be addressed:

- Split/join separation is necessary unless we enumerate all combinations in block structures and introduce BPEL-style links or a similar free-graph technique.
- Separation should also go for *Multiple Instance* patterns.
- *Synchronizing Merge* remains a problematic but necessary pattern. It must be given a formal semantics; and should preference should be given to a semantics that will scale to distributed settings.
- All patterns should have a *formal* description. Overlaps should be reduced as much as possible.
- Some patterns—e.g. *Interleaving* and *Milestone*—could be understood better if perceived as data patterns.
- Simple predicates on splits and joins would could do a lot of the work needed (cf. EPCs or BPEL XPath Join Conditions).

If in addition we desire a workflow language that is simple, unbiased, and amenable to analysis, we must identify the shortcomings of current standards and models and seek out ways alleviate those. To that end, we have established that:

- EPCs are flexible but still need a formal semantics.
- Only allowing block structure—e.g. XLang—in insufficient. It is provably less expressive [34].
- BPEL circumvents this through links, but as a model lacks proper support for multiple instances. BPEL also lacks a concisely described semantics, and imposes serious constraints on free-graph idioms (links cannot cross certain blocks etc.).
- The π-Calculus and CCS can easily serve as foundation. Like Petri nets they have serious shortcomings for cancellation patterns. In distributed workflow systems, where a solicited task can be accepted a several different peers, the π-Calculus and CCS suffer from the consensus problem—an arbiter is needed to decide who actually gets the task. As long as there is only one centralized workflow server, this is not a problem. The system may experience scalability issues because the acceptance of one task causes the removal of other tasks, and this must happen atomically. Also see section 3.6 for more on solicitation and allocation patterns, also known as resource patterns.
- Petri nets are good de facto workflow modeling, but they lack flexible constructs for generativity (cannot copy nets). Whereas one has to concede that Petri nets work surprisingly well for free-form graphical structures, the introduction of advanced cancellation, exceptions, and distributed failure strongly suggests a less graphical approach. Petri nets do not handle cancellation and exceptions well—if at all—at this point [69], and complex cancellation/failure scenarios often do not have a pleasant graphical representation. More importantly, the workflow designer should not be limited because of graphical restrictions.
- A formal framework for expressiveness should be put in place. E.g. combining bisimulation notions with Felleisen's characterization of expressiveness[17][21]. This should lead to a partial order on the expressiveness of workflow models.

[21] Quoting Felleisen [17]: ... "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.

The CCS encoding of the control flow patterns and the subsequent pre-design of SMAWL provided a response to van der Aalst's challenges 4–7 to (4) model the Petri net shown in figure 3.1, (6) model existing patterns, (5) provide new challenges, and (7) suggest new patterns (especially using mobility).

We expressed all patterns and several variants, e.g. we can combine deferred and non-deferred choices (it is not clear if this is possible using the current patterns). It would be interesting to see if Petri nets can handle the overlaying operator more elegantly than CCS. Also, a practical investigation into applicability of the overlaying operator might be very fruitful. In remains open if the newly discovered patterns occur often enough in practical settings to warrant their incorporation with the rest. Real world examples to support them would be very valuable.

The challenges proposed π-calculus, but mobility was not needed. 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[22]. 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. 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. It would seem that the π-calculus notion of mobility would be convenient for modeling service infrastructure and implementation of actual workflow tasks. For now this means that control flows are simpler and therefore easier to analyze.

When debating expressiveness an easy, but vacuous point is Turing completeness. Indeed, the π-Calculus is Turing complete, but in its raw form it is unsuited for any non-trivial programming/modeling task. For domain-specific languages like the ones debated here, it would seem more reasonable to consider criteria such as (a) ontological fit (i.e. domain closeness, convenience of expressing common idioms), (b) expressiveness, (c) amenability to formal analysis, (c) simplicity, and, for workflow languages, (d) graphical representation.

It remains extremely important to provide an intuitive graphical user interface—when possible—and easily understandable high-level abstractions for the user, and this should be addressed in future work. SMAWL took the first speculative step in the direction of a high-level control flow language, but SMAWL still lacks a graphical user interface (auto-generation and editing) and a base language for data manipulation. The most significant shortcoming of SMAWL it semantics. SMAWL semantics are currently defined in terms of the map $\mathcal{T}[\![\cdot]\!]$, but we have no guarantee that this map is correct (correct with respect to what?) or that SMAWL actually covers the patterns amply. The formal pattern description and expressiveness framework sought above will help accommodate this in the future.

SMAWL was designed with two goals in mind: covering all patterns and providing high-level constructs. These goals were met, although as pointed out, eliminating

[22] Brief studies suggest that the Join calculus [19] is syntactically more convenient because it emphasizes atomic joins. The problems with distribution remain, though.

signal passing however nice was not possible. SMAWL being on syntactic sugar on CCS retains the properties of CCS in terms of analysis. Some more information is available because an analysis can draw on both the SMAWL program and the derived CCS expressions.

SMAWL misses a graphical representation, which could be an adapted version of UML 2.0 Activity Diagrams. As demonstrated by Wohed et al. [79] UML 2.0 Activity Diagrams are able to express almost all workflow control patterns[23]. SMAWL still needs a companion data language, it needs distribution, and possibly code mobility and locality. It is highly likely, that SMAWL would fall short for purposes of process mining, simply because the constructs are so chunky. This is an open question.

It is noteworthy that we have been able to make do so far without the notions of mobility or locality found in π-Calculus and Mobile Ambients [10] or Bigraphs[31], respectively. Furthermore, as noted, we have been able to accept the artificial global consensus inherent in the π-Calculus and CCS, simple because we worked with the assumption of a centralized workflow engine. If we loosen this asumption, models like Actors [2] or the recently proposed Transactors [18] may prove more fit.

In conclusion the four major research issues to be addressed today are:

1. Formalizing and simplifying the control flow patterns.
2. Analyzing the issues facing distribution, transactions, and cancellation patterns.
3. Formally developing an appropriate calculus, and, further down the line, a language.
4. Implementing a distributed workflow management system based hereon.

3.6 Data, Resource, and Service Interaction Patterns

In the wake of the control flow patterns, data patterns [56] and resource patterns [57] were proposed. Thus far 39 data patterns are documented, spanning the areas of scope (*visibility*), data sharing (*interaction*), data passing (*data transfer*), and interfacing with control flow (*data-base routing*). The data patterns contain few controversial ideas, but they do seem slanted towards an imperative paradigm where shared data can be updated. The paper [56] analyzes BPEL4WS and XPDL with respect to the data patterns.

Resource patterns describe the mechanism of delegating enabled tasks in workflow to willing and able agents for execution. They include push and pull patterns, round robin allocation, history-based allocation, clustered allocation, role-based allocation, suspension/resumption, and several others. Resource patterns are implemented as delegation layer between the workflow engine soliciting tasks and the agents executing tasks. The most likely model is a state machine representing the state of each task, but this imposes new requirements on the interface to tasks (e.g. they must be able to receive suspend/resume/cancel messages). The setup has a certain similarity to auction mechanisms, and thus it would be interesting to consider resource patterns from a game-theoretic perspective (offering tasks to the highest bidder etc.).

[23] Workarounds using signal-passing are needed for Interleaved Parallel Routing and Milestone. MI without a priori Runtime Knowledge is not supported and there are serious deficiencies when it comes to the ever-illusive Synchronizing Merge. Activity Diagrams are significantly weaker when it comes to data patterns where about half of the 40 patterns are covered.

In what seems to be an accelerating race for more and more patterns, the notion of *Service Interaction Patterns* has been proposed [3]—ostensibly to demonstrate the justification of π-calculus for workflows as prompted by van der Aalst. The content of this contribution will be investigated later on.

Chapter 4

A Model for Business Data

This chapter is about business data from an accounting perspective. It reflects a substantial amount of time that has gone into the analysis of the Resources/Events/Agents model (REA). We do not delve into a critique of REA since this has already been done in Signe Ellegård Borch and Christian Stefansen "Evaluating the REA Enterprise Ontology from an Operational Perspective" [9]. Instead, we give an introduction and formalization of traditional double-entry bookkeeping and a short introduction to the REA model.

Previously, we claimed that business data and business processes are orthogonal aspects of an enterprise system. This chapters presents the traditional model and the heretical model, both of which can be used in conjunction with a process language to form the basis of an enterprise system.

4.1 Double-Entry Bookkeeping

Ontologically it is easy to identify *assets* as the *scarce resources of utility at the disposal of the firm*. *Liabilities* are also easily identified as *scarce resources of utility that the firm owes to others*. We can keep the accounts by tracking only assets and liabilities[1], but we can then make no statements about the sum of the amounts in a transaction. Some transaction will affect assets only, some liabilities only, and some both. The system lacks a sanity check and is prone to errors.

Clearly the value of the firm to its owners, *owner's equity*, can be computed at any time as the function:

$$Owner's\,Equity = Assets - Liabilities$$

The epiphany that lies in the heart of double-entry bookkeeping is to *track owner's equity explicitly*. This gives rise to the fundamental invariant

$$Assets = Liabilities + Owner's\,Equity$$

[1]Danish readers should notice that assets are *aktiver*, and the sum of liabilities and owner's equity is *passiver*. Owner's equity could translated as *egenkapital*, and liabilities as *fremmedkapital* or simply *gæld*.

widely known as *The Accounting Equation*. In a common textbook explanation, the left side explains *what* resources are owned, the right side explains *who* owns them. To maintain the invariant dictated by the Accounting Equation every transaction must obey the invariant

$$\Delta Assets - \Delta(Liabilities + Owner's\, Equity) = 0.$$

For additional convenience we negate $Liabilities + Owner's\, Equity$ to obtain

$$\Delta Assets + \Delta(-Liabilities - Owner's\, Equity) = 0$$

so that every transaction always has both a negative and a positive component regardless of the parts it affects. It the traditional jargon these components are called *debit* and *credit*, and with the aid of these labels only positive numbers are needed.

4.1.1 Going formal

In its most simple form the ledger has a chart of accounts and a transaction list. These are straightforward to express:

Definition 1 *In the chart of accounts an account is described by a unique identifier $a \in I_a$ and some associated properties (data) $d_a \in \mathcal{D}_a$. Thus the chart of accounts $\mathcal{A}$ can be defined as a finite set*

$$\mathcal{A} \subseteq I_a \times \mathcal{D}_a$$

where given two accounts $(a_1, d_1), (a_2, d_2) \in \mathcal{A}$ we have $a_1 = a_2 \implies d_1 = d_2$.

Definition 2 *A transaction needs an identifier $i \in I_t$, a time stamp[2] $t \in \mathbb{N}$, an account identifier $a \in I_a$, some associated data $d_t \in \mathcal{D}_t$ and a currency amount $c \in \mathbb{Z}_{/k} \setminus 0$, where we use $\mathbb{Z}_{/k} = \{z : z \cdot k \in \mathbb{Z}\}$ to denote fixed precision monetary amounts, i.e. $k = b^r$ gives an r decimal precision in base b.*

An transaction entry *$e_i$is a tuple*

$$(i, t, a, d, c) \in I_t \times \mathbb{N} \times I_a \times \mathcal{D}_t \times \mathbb{Z}_{/k} \setminus 0.$$

The transaction identifier i_a (most often a number and most often numbered successively in order of entry) is used to mark a number of entries as part of one transaction. If there is any physical evidence of the transaction, such as requisitions, invoices etc. they will be marked with the same identifier. For reasons of storage and retrieval there will often (read: always) be a (total) ordering on I_t.

Definition 3 *A* transaction *t_i is a finite, non-empty set of entries that all have the same transaction identifier, i.e. for all $(i_j, t_j, a_j, d_j, c_j), (i_k, t_k, a_k, d_k, c_k) \in t_i$ we have $i_j = i_k$.*

If we lay out the transactions in ascending order of transaction identifier i_a we get the transaction trace of the ledger.

[2] Strictly speaking the time stamp could be made part of the associated data, but since it is unconceivable to have a system without it, we put it here for convenience.

Definition 4 *A pair $(\mathcal{E}, \mathcal{A})$, where $\mathcal{E}$ is a finite set of entries e_i and $\mathcal{A}$ is a chart of accounts, is called a* ledger *if and only if the following conditions are satisfied:*

Transaction invariant *For all transactions $t_i \subseteq \mathcal{E}$*

$$\sum_{(i,t,a,d,c)\in t_i} c = 0.$$

Balance invariant $\sum_{(i,t,a,d,c)\in\mathcal{E}} c = 0$

Atomic transaction time stamp *For all transactions $t_i \subseteq \mathcal{T}$ the time stamp of each entry is the same, that is for all $(i_j, t_j, a_j, d_j, c_j), (i_k, t_k, a_k, d_k, c_k) \in t_i : ts_j = ts_k$.*

In order to comply with generally accepted accounting principles, additional rules must be observed. Transaction should be kept as small as possible, i.e. we cannot just put everything together in one transaction. It is tempting to say, the transaction grouping of entries should be the most fine-grained of the groupings possible. This is close, but not completely in keeping with practice. Also accounts should be classified at the very least as being *assets*, *liabilities/owner's equity*, *income* or *cost* accounts. This basic method works fine without these classifications, though.

The avid reader will have noticed that the global *balance invariant* follows directly from the local *transaction invariant*. Furthermore, transactions are always added in ascending order of transactions identifiers, and since we can never remove a transaction, once it has been added, the ledger grows monotonically with the transaction identifiers. Because of this property it is always possible to find out what the state of the firm was up to a given transaction number. Common best practice prescribes that the bookkeeping should happen continuously and for this reason we are unlikely to see significant jumps in timestamps when looking through the transactions in order.

Also notice that the system lacks an undo operation. While this could be considered a very serious deficiency, it is not because of a serendipitous other property: All transactions have an inverse transaction that undoes them. That is, given a transaction $T = \{e_1, \ldots, e_n\}$ we require a transaction $T^{-1} = \{e_1^{-1}, \ldots, e_n^{-1}\}$ so that for each $e_i = (i, t, a, d, c) \in T$ and $e_i^{-1} = (i^{-1}, t^{-1}, a^{-1}, d^{-1}, c^{-1})$ we have $a = a^{-1}$ and $c = -c^{-1}$. Informally, we post the negated amount on each of the accounts. (Technically, we do not care about the order of the e_i^{-1}, as long as there exists a pairing of the e_i with the e_j^{-1}.) We can see that T and T^1 are inverse in the respect that for each account identifier $a \in I_a$ we have

$$\sum_{(i,t,a',d,c)\in T\cup T^{-1}} [a = a'] \cdot c = 0$$

—the combined effect on any account is zero. The notation $[p]$ denotes the function that maps to 1 if the predicate p is true and 0 otherwise.

All this means that not only does the system have a natural way of undoing or modifying previous transactions, it also retains a record of this having occurred.

In summary the principle of double-entry bookkeeping possesses some highly desirable properties:

1. It elegantly maintains a local (transaction) invariant that results in a global (balance) invariant.
2. The ledger is monotonic in the transaction identifiers.
3. It is simple.
4. All transactions have an inverse.

A traditional ledger also has the following disadvantages:

1. It records only a particular subset of economic data.
2. It records economic data in a aggregated form e.g. only the grand total of an invoice is recorded.
3. It has no notion of process. Even if some accounts (e.g. accounts receivable) represent a non-accepting process state, there is no explicit representation of the process. The system is likely to build up residues, i.e. orphan amounts sitting indefinitely in account that represent a non-accepting states. Cleaning up residues can is often tedious and requires additional process information not available in the ledger.

Items (1) and (2) can be fixed relatively easy through use of the data fields d, which happens to the solution that many current providers have opted for. Item (3) should come as no surprise, since we are dealing with a data model for transaction traces. Whatever data model we pick, the system must be brought to bear on both a data model and a process model, like the ones discussed in the previous chapters.

It does seem, though, that the data model could be more accommodating for registration of non-economical data. Of course the basic format of the tuples could be changed. Transaction identifiers, amounts, and account references could all be packaged in the data field yielding a simple format of (t, d)—time stamp and data. Even the time stamp could be taken out, if we have some ordering on the entries to form a trace. Even if it is possible to reduce the entries of a trace to just one generic data field, it will be desirable for the most to have a more detailed basic data model. We have seen that the double-entry bookkeeping data model can serve this purpose, but it has no natural connection with the notion of agents working on resources in a workflow system. We will now consider another model that is ontologically closer to these notions.

4.2 Resources/Events/Agents (REA)

The Resources/Events/Agents model (REA), like double-entry bookkeeping, is an accounting model. The data model we are seeking need not be an accounting model *per se* to serve our purposes—other enterprise ontologies will do—but we do need to argue that the data model we choose can express financial data in accordance with generally accepted principles. With REA, being an accounting model by design, this part comes for free.

The REA model [41], provides a foundational ontological pattern for accounting systems: A **R**esource is affected in an **E**vent carried out by an **A**gent. Usually there will be an initiating agent and a receiving agent. The microeconomic basis is the theory of the firm, and hence every event—at least theoretically—has a *dual* such that two (or

perhaps more) events together constitute an *exchange* in the microeconomic sense of the word (i.e. an exchange of *scarce resources*).

4.2.1 Core REA

Core REA is taken to mean the definitions of *Resources*, *Events*, *Agents*, and *duality*. Basic REA as introduced in the 1982 paper also mentions *economics units*, *macro-duals*, etc. Let us first visit McCarthy's definitions of the core entities and then discuss the merits of the Core REA as an accounting and data model.

Definition 5 (Economic resources) *Economic resource are defined by Ijiri [1975, pp.~51-2] to be objects that (1) are scarce and have utility and (2) are under the control of an enterprise. In practice, the definition of this entity set can be considered equivalent to that given the term "asset" by the FASB [1979, pp.~51-7] with one exception: economic resources in the schema do not automatically include claims such as accounts-receivable. [41]*

Definition 6 (Economic events) *Economic events are defined by Yu [1976, p.~256] as "a class of phenomena which reflects changes in scarce means [economic resources] resulting from production, exchange, consumption, and distribution." [41]*

Definition 7 (Economic agents) *That is, they are identifiable parties with discretionary power to use or dispose of economic resources. [41]*

Every observable transaction is registered as a tuple (r, e, a, a). Thus as transactions occur we build up a trace of (r, e, a, a) tuples (often just sloppily referred to as "events") to describe the history and state of the enterprise.

4.2.2 Duality

Example 8 *Customer A purchases three different products X, Y, and Z on mutually different dates. Customer A pays in two installments on two different dates. Neither of the installments match the exact amounts of the products sales. There are five events: three sales events and two payment events.*

As said, REA is based on the notion of *exchanges*, which are pairings of events. This relationship between events is known as *duality.* In [41] duality is described as connecting "each increment in the resource set of the enterprise with a corresponding decrement"[3]. Thus duality is a bipartite graph, where (r, e, a, a) tuples are the nodes, and the edges are (parts of) duality relations. In REA every event should eventually be paired with one or more other events in duality. This makes it easy to distinguish

[3]The hinted one-to-one nature of this relationship should not be taken too literally; otherwise a common scenario such as example 8 cannot be modeled without aggregating or disaggregating beyond observable facts, i.e. allocating the cash payments to the three sales events by splitting them into more events or merging the payments and the sales into two event.

outstanding claims (events where no dual has been registered yet) from completed exchanges.

Duality is representative of the mindset the REA model tries to install in its users. For every cost, one must ask: why *exactly* do we incur this cost? To answer this question we must point to the processes that require it and thus allocate it. If multiple reasons are identified, the cost should be allocated accordingly. This is very similar to the principles of Activity-Based Costing [7]. If no reason can be identified at all, the cost should be eliminated. We may need an exception on this when it comes to taxes. Identifying the resources involuntarily "bought" through taxation, is far too involved for practical applications. Instead the forced dual constraint is loosened in such cases. This is referred to as *implementation compromises*. Implementation compromises are described in "Augmented Intensional Reasoning in Knowledge-based Accounting Systems" p. 12 [24] and they are a necessary evil in any REA implementation. The necessity of implementation compromises in an operational ontology is undesirable for reasons of interoperability, automated reasoning, and consistency—and simply because this is symptomatic of a badly designed ontology. This critique is elaborated in [9].

For accounting-savvy readers[4] it is important to notice that the duality relation and the double-entries found in traditional $A = L + OE$ bookkeeping are not the same, bar in some special cases such as cash purchase.

4.2.3 Issues

The academic papers on REA say very little about the attributes or format of the components in a (r, e, a, a) tuple. As with double-entry bookkeeping, we can just consider them data fields to be defined and types at our discretion. This freedom, while convenient for our purposes, has been the target of substantial critique pointing out that with the current amount of generalization, REA is not an interoperable enterprise ontology.

Most likely an actual system will extend resources, events, and agents with types, hierarchy, and aggregation. McCarthy seems to suggest adopted a static description for these structures in saying that "[t]he only static object in the generalized schema concerns responsibility; . . . " [41], but this introduces problems of maintaining accurate history when from time to time static hierarchies must be changed. A similar problem arises in double-entry bookkeeping, when modifying the chart of accounts.

If one decides to use REA for an enterprise system, it should be clear that it will make the mark when it comes to financial reporting. Mapping the double-entry ledger to REA is reasonably straightforward, bar one thing: owner's equity. Modeling owner's equity explicitly was the epiphany that lead us to double-entry bookkeeping. McCarthy claimed that owner's equity is an accounting artifact, and thus made a point not to include it as an explicit part of the REA model. It is unclear how to express accruals, dividends, capital adjustments, and more arcane equity distributions in REA, and this *must* be sorted out before REA can harbor any hopes of becoming an accounting model to be reckoned with.

[4] Id est, readers who have digested the previous section.

Materialization—the process of generating an income statement, balance, cash-flow statement etc.—is left as an implementation problem for system designers. A better approach would be a reference implementation or a handbook in REA accounting. There is a serious number of issues pertaining to the latest extensions of the REA model, such as *commitments*, *policies*, *the contract state machine*, and *types*. We feel that these have not been adequately described to warrant further investigation and opt for a process model based on concurrency research as sketched in previous chapters.

When implementing REA one should remember that REA is a meta-model meaning that meta-programming[5] is needed. REA needs a notion of types for agent, events, and resources, and the model becomes more involved when users are allowed to define their own types and instances.

4.2.4 Related Work

The REA model can be and has been examined from many angles of research. Previous research on REA has focused primarily on modeling aspects—in particular the REA model as an ontological[6] framework for ERP systems [22]. This research has unveiled a number of idiosyncrasies in the REA model, but a formal approach remains an untried path even though the model would benefit immensely from such groundwork—as one analysis by Jaquet clearly indicates [30].

Later extensions [24, 22, 23, 21] aimed to make REA a full-blown enterprise ontology, their most important contribution being the notions of *types*, *commitments*, and *policies*. A fair number of follow-ups have been published: Jaquet [30] advocates an global (independent) perspective as opposed to REA's original trading-partner perspective and suggests corrections to the *stock/flow*, *inside/outside*, *cooperation*, *partner*, and *assignment* relations; Jaquet and Lampe [30, 38] independently find the REA model ontologically unsound with respect to Gruber's ontology criteria[7] [26] and Sowa's classification [61]; David [15] suggests a dichotomy of *economic* events and *information* events; and the UN/CEFACT Modeling Methodology standard [66] incorporates foundational REA. Contending enterprise ontologies exist (e.g. TOVE, Enterprise), but none of these have the strong accounting heritage of REA.

4.3 Conclusions and Future Work

The idea of resources, events, and agents is immediately believable, and it fits well with the idea of a process-based programming language that generates event traces in reaction to other events. Everything else in the REA model must be argued more thoroughly.

We now have two data models which could both serve as the foundation in an enterprise system. Having two, where only one is needed, will make us more careful

[5]Meta-programming is writing code that generates, modifies or inspects other code. Cf. MetaML, ModalML, FreshML

[6]The word ontology can be read as "a explicit specification of a conceptualization" (Gruber 1993 [26]). In an even more plain explanation it is simply "a vocabulary".

[7](1) Clarity, (2) Coherence, (3) Extendibility (4) Minimum encoding bias (5) Minimal ontological commitment

not to tie the process model and the data model too tightly. It should be possible to change data model at will.

The REA model needs more research—or rather clarification. It needs to be properly defined, its weaknesses must be identified and assessed, and its purposes more clearly outspoken. We have scrutinized commitments, types, policies and contracts only to find that not even the informal semantics were clear. It seems the type system could be that of Odell's Power Types [48] (which in turn also need an overhaul), commitments and contracts can be captured nicely within a process language, and policies could be rules in, say, LTL on the types and instances of the basic three entities and duality. The REA model may have the potential to do more than just serve as a rudimentary data model, but this requires a more focused research effort—and this remains future work[8].

[8] ...but not for me.

Chapter 5

Toward a Declarative Framework

In the previous chapters we have considered choreography, orchestration/processes, and data models. We are yet to consider a fourth component, namely that of business rules. The ultimate goal is to make these components fit together to form a framework for enterprise systems. As mentioned previously there are two approaches: (1) building process layers on top of the existing legacy systems by using service interfaces or (2) rebuilding the enterprise system using a process-oriented language. Our main interest is option (2), but as time goes by, more and more legacy code will have been recoded in the new languages, and the difference between the two solutions will become smaller. There will always be legacy code and auxiliary services, but, in fact, these will just be treated similarly to the services invoked on other peers[1]. Apart from their service description, we know nothing about them. Hence the mission is to get the process-oriented framework in place, and start expanding its scope afterwards.

We need a road map to get to this framework:

1. Identify and assess requirements
2. Consider current industry standards w.r.t. requirements
3. Consider current formal models w.r.t. requirements
4. Design candidate process language, data model, base language, and report language
5. Develop formally and implement[2]

[1] We should be careful though! As Waldo et al [77] point out, such a unified view of local and distributed objects may be fallacious:

> The major differences between local and distributed computing concern the areas of latency, memory access, partial failure, and concurrency. The difference in latency is the most obvious, but in many ways is the least fundamental. The often overlooked differences concerning memory access, partial failure, and concurrency are far more difficult to explain away, and the differences concerning partial failure and concurrency make unifying the local and remote computing models impossible without making unacceptable compromises.

These observations are equally relevant to business application programming in a service-oriented setting, but the conclusions may differ.

[2] "To program is to understand"

Chapter 3 presented a first iteration over items (1), (2 - partially), (3 - partially), and (4) for the process language, but this was in a centralized setting that needs to be extended to the distributed setting. Likewise items (1)-(5) must be addressed for the data model, the base language, and the report language. Some of this is ongoing work in the NEXT research group. At the heart of all this lies the architecture that binds everything together.

5.1 Architecture

We envision an architecture built on three core notions:

Processes Processes are responsible for executing business logic with the help of other processes local or remote in reaction to incoming events from either other systems, users or a timer.

Log Every observable transition in a process is logged and thus the log is the data structure that describes the history of the system (equivalently the state of the processes describe the possible futures, and the processes together with the log describe the entire system state).

Reports All financial reporting can be carried out by reports: small domain specific programs that generate and maintain reports about system state based on the log and, possibly, the running processes. Some reports are just predicates that can raise an flag to warn of a condition, others maintain financial data, other maintain base data etc.

The processes invoke services to get work done. Sometimes they push the work, other times the work is pulled upon solicitation. This is handled by a delegation layer, a meta-process, which provides support for common resource patterns. The processes themselves are continuously analyzed by meta-processes. These attempt to detect deadlocks, dead-paths etc. and generally provide the runtime service to the processes. Contrary to the business processes, which are written in our DSL, meta-processes are probably written in a full programming language. Data mining meta-processes can examine the log continuously to suggest improvements and identify weakness in current processes.

Both of the investigated data models have a desirable monotonicity property that will make them fit well into this architecture. The absence of static data has several conveniences: the system keeps a complete history of all events, so even if base data is changed, the system can still revert to a previous state—it always maintains a complete history. This also means that base data, say the address of a customer, can be changed in the future simple by adding a process that triggers the update event on a particular date. Of course there are serious performance penalties in such a design. Drilling through the log to find the address of a customer every time it is needed is clearly not desirable. Monitors can keep their own state, such as a list of customers and their current balance. The reports simple observe the log and take care that update events are reflected in their result. The technique of finite differencing [50] has been suggested for this purpose and will work for a subset of reports that use a restricted language and operate on the log only. Finite differencing transforms a report on the entire log into

its "derivative" report that computes the new report based on the previous report, the newly arrived events, and some auxiliary data. Transactions in this framework amount to building a separate log for the transaction and then either dropping it or merging it with the mother log on rollback/commit.

There are other means of alleviating the performance issues: After some time, maybe a year, a snapshot can be created and the log can be truncated beyond a half year time horizon. Reports can be push- or pull-based. Only critical reports should employ push—others can use timed polling or a similar strategy. Also, reports should supply a filter or predicate specifying what events they are interested in. This will greatly reduce the number of reports that need to examine each event, but it raises the interesting problem of deciding each of the report filters for each event in a computationally efficient manner. A framework called Complex Event Processing [39] has been proposed to this end and several database stream query languages exist [25].

There are of course things that the system does not record in the log unless specifically built to do so. Most importantly the doings of the meta-processes, i.e. (1) creation/modification of reports, (2) creation/modification of processes. These are minor issues, though as this will rarely be the subject of reporting. If we desire to replay or simulate a simple meta-log will do.

Similar process-/event-based ideas have been sketched before, including a Petri net-based framework [14] and a π-Calculus-based framework with an emphasis on behavior types[42].

5.2 Requirements, Standards and Models

As outlined in the road map the first step toward the design of the languages in our declarative framework is—not very surprisingly—requirements. For now, we have a number of patterns that—after weeding out the worst ones—will serve as a good indicator of the expressiveness needed. Other requirements will become apparent as current standards and concurrency models are examined and compared.

A staggering number of industry standards exists in the context of SOA. These cover meta-formats (XML), addressing, discovery (UDDI), security (WS-Security, WS-Trust, SAML (Security Assertion Markup Language), XACML (eXtensible Access Control Markup Language)), messaging (SOAP, WS-ReliableMessaging), description (WSDL), orchestration (WS-BPEL, XLang, WSFL, BPML), notation (BPMN), choreography (ebBPSS, WS-CDL, WSCI, WSCL) and much more. The interested reader is referred to Cutler and Denning [13] for a brief informal February 2004 snapshot of web service related standards. In addition one can always monitor the stakeholders, namely OASIS, W3, BPMI, Microsoft, IBM, BEA, SAP, Oracle etc. Much of this work refers only to web services, although the ideas are more widely applicable. Indeed many standards are simply a re-bottling of textbook ideas from distributed systems research.

Similarly, on the research side there has been an astounding number of attempts to come up with a universal programming platform for distributed (business) applications. The most mature and well-known models are those which we have used here, namely Petri nets and π-Calculus, but many others exist to introduce additional concerns: Join calculus (scoping and join), LOTOS (Language of Temporal Ordering

Specifications)[46, 8] (combines process specification and algebraic specification), Spi (encryption), Orc, Mobile Ambients, Boxed Ambients (locality), Bigraphs (unifying theory of concurrency), Actors, Transactors (distributed data consistency).

In addition a number of type systems have been suggested: linear/affine types [36, 37], sorts, information flow types [28], etc. Lately, Poly*, a type system *scheme* for mobility calculi was suggested by Makholm and Wells [40].

Although large numbers of contributions exist on both sides, it is probably fair to say that the cross-fertilization has been somewhat lacking. Some formal model have resulted in actual programming languages, most notably Pict [52], TyCo[3], Concurrent ML[4], and Join [19]. In the work ahead, the individual merits of each of the important standards, models, and language available should be assessed. So far we have seen some clear merits of the π-Calculus: it is compositional, can express non-determinism, has an explicit representation of state, and can be synchronous or asynchronous. These properties will still be needed. The π-Calculus also has its oddities, e.g. several "peers" can listen on the same channel creating a non-local consensus problem, and it cannot model three peers a, b, and c with private channels between each pair, because of its inherent syntactic tree structure. Furthermore, it is likely that locality will play a larger role in the future.

5.3 Business Rules and Verification

So far we have not touched upon the area of writing business rules and verifying (statically or dynamically) that processes satisfy them. A broad spectrum of methods is available ranging from theorem proving over software model checking to type checking of behavior types. Some current ideas pending further investigation include using *aspects* for runtime verification and using finite differencing of functional predicates for runtime verification.

> "Formal methods will never have any impact until they can be used by people that don't understand them." — (attributed to) Tom Melham

[3] http://www.ncc.up.pt/tyco/
[4] http://cml.cs.uchicago.edu

Chapter 6

Conclusions and Future Work

Let us return to the main design objective:

> *A declarative programming platform for writing enterprise systems and automating business processes in a service-oriented architecture.*

As argued such a platform can be thought of as having three interconnected aspects, namely a data model, a process model and a reporting model. Work remains in each of these areas, but the focus is on a process model first and then—time permitting—the reporting model. The data model so far seems the least important part. Choreography, while interesting, is not crucial to progress.

The process model is an accessible area. Many models and standards exist, and the catalogs of workflow patterns are convenient benchmarks. As pointed out, the workflow patterns could need a more formal presentation, but this is not necessarily a path leading to a better process language in the end. As things are now, the obvious ambiguities and lacking definitions spark interesting discussions and promote awareness of the complexities of such languages. Formalizing the workflow patterns would certainly help; the questions is whether it is strictly necessary. A much more important issue is that of distribution. BPEL, YAWL, π-Calculus, and Petri nets all have serious limitations, and new models such as Transactors seem promising. The path of research from here should be the one leading to a set of objective criteria for a process language; this path may or may not involve a formalization of workflow patterns. The criteria in place, models can be compared, a new model can be developed if needed, and the model can be formally tested and deployed.

Once the process language is somewhat stable, we can address reporting, monitoring, analysis etc. Performance plays a great part here, and so an implementation will be required. Furthermore, a running prototype opens up to more experiments: (1) Automatically generated user interfaces based on process expressions, (2) event log mining (process mining) to recover processes, (3) resource pattern designs, and (4) process analysis and verification.

The road map presented in conclusions of chapter 3 remains the most important direction of research: The number one objective is to produce a process model and then establish a prototype framework.

Appendix A

Publications

(J = Journal article, C = Conference paper, S = Short conference paper, W = Workshop paper, TR = Technical report, P = Presentation (paper presentations not included), T = Teaching)

J1 J. Andersen, E. Elsborg, F. Henglein, J.G. Simonsen, C. Stefansen, "Compositional Specification of Commercial Contracts". International Journal on Software Tools for Technology Transfer (STTT), Springer Verlag, to appear.

C1 J. Andersen, E. Elsborg, F. Henglein, J.G. Simonsen, C. Stefansen, "Compositional Specification of Commercial Contracts". Proceedings of the 1st International Symposium on Leveraging Applications of Formal Methods (ISOLA 2004). University of Cyprus Report TR-2004-6, 103-110. University of Cyprus 2005.

S1 Christian Stefansen, “SMAWL: A SMAll Workflow Language Based on CCS”. Proceedings of the 17th Conference on Advanced Information Systems Engineering Forum (CAiSE 2005 Forum), University of Porto 2005.

W1 Signe Ellegård Borch, Christian Stefansen, "Evaluating the REA Enterprise Ontology from an Operational Perspective". Enterprise Modelling and Ontologies for Interoperability, EMOI - INTEROP 2004, CAiSE Workshops (3) 2004: 144-152.

—

TR2 Christian Stefansen, “SMAWL: A SMAll Workflow Language Based on CCS”. Harvard University Computer Science Technical Report TR-06-05, Harvard University 2005.

TR1 J. Andersen, E. Elsborg, F. Henglein, J.G. Simonsen, C. Stefansen, "Compositional Specification of Commercial Contracts". DIKU Technical Report 05-04. University of Copenhagen 2005, to be printed.

—

P2 Christian Stefansen, “Formal Commercial Contracts”. Presentation given June 2004 at Microsoft Research, Redmond, WA.

P1 Christian Stefansen, "Workflow Modeling Using the pi-Calculus". NEXT presentation given June 2003 at ITU.

T1 Fritz Henglein, Christian Stefansen, "Miniseminar: Introduktion til Resources/Events/Agents-modellen". 2 ECTS course given August 2003 at DIKU.

Bibliography

[1] W. Aalst, M. Dumas, A. Hofstede, and P. Wohed. Analysis of web services composition languages: The case of BPEL4WS. In *Proc. of ER'03*, volume 2813 of *Lecture Notes in Computer Science (LNCS)*, pages 200–215. Spinger, 2003.

[2] Gul A. Agha, Ian A. Mason, Scott F. Smith, and Carolyn L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7(1):1–69, January 1997.

[3] Marlon Dumas Alistair Barros and Arthur ter Hofstede. Service interaction patterns: Towards a reference framework for service-based business process interconnection. Technical Report FIT-TR-2005-02, Faculty of Information Technology, Queensland University of Technology, Brisbane, March 2005.

[4] Jesper Andersen, Ebbe Elsborg, Fritz Henglein, Jakob Grue Simonsen, and Christian Stefansen. Compositional specification of commercial contracts. Technical Report 05-04, DIKU, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen, Denmark, July 2004. http://topps.diku.dk/next/contracts.

[5] Jesper Andersen, Ebbe Elsborg, Fritz Henglein, Jakob Grue Simonsen, and Christian Stefansen. Compositional specification of commercial contracts. In Bernhard Steffen, editor, *Proceedings of the 1st International Symposium on Leveraging Applications of Formal Methods*. University of Cyprus, November 2004.

[6] Tony Andrews, Shaz Qadeer, Sriram K. Rajamani, and Yichen Xie. Zing: Exploiting program structure for model checking concurrent software. In Philippa Gardner and Nobuko Yoshida, editors, *CONCUR*, volume 3170 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2004.

[7] Anthony A. Atkinson, Rajiv D. Banker, Robert S. Kaplan, and S. Mark Young. *Management Accounting*. Prentice Hall, third internation edition edition, 2001.

[8] Tommaso Bolognesi and Ed Brinksma. Introduction to the iso specification language lotos. *Comput. Netw. ISDN Syst.*, 14(1):25–59, 1987.

[9] Signe Ellegård Borch and Christian Stefansen. Evaluating the REA enterprise ontology from an operational perspective. In *Proceedings from the Open INTEROP Workshop on "Enterprise Modelling and Ontologies for Interoperability", EMOI - INTEROP*, June 2004.

[10] Luca Cardelli and Andrew D. Gordon. Mobile ambients. In *Foundations of Software Science and Computation Structures: First International Conference, FOSSACS '98*. Springer-Verlag, Berlin Germany, 1998.

[11] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite state concurrent system using temporal logic specifications: a practical approach.

In *POPL '83: Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 117–126. ACM Press, 1983.

[12] Francisco Curbera, Rania Khalaf, Frank Leymann, and Sanjiva Weerawarana. Exception handling in the bpel4ws language. In Wil M. P. van der Aalst, Arthur H. M. ter Hofstede, and Mathias Weske, editors, *Business Process Management*, volume 2678 of *Lecture Notes in Computer Science*, pages 276–290. Springer, 2003.

[13] Roger Cutler and Paul Denning. Annotated list of web services specs, February 2004.

[14] Nikunj P. Dalal, Manjunath Kamath, William J. Kolarik, and Eswar Sivaraman. Toward an integrated framework for modeling enterprise processes. *Communications of the ACM*, 47(3):83–87, 2004.

[15] Julie Smith David. Three "events" that define an REA methodology for systems analysis, design, and implementation.

[16] Yang Dong and Zhang Shensheng. Modeling workflow patterns with pi-calculus. Unpublished. http://www.workflow-research.de. Available from http://www.workflow-research.de/Forums/index.php?s=256f7df1bf34ea7cd72bc8af62e9ea96-&act=Attach&type=post&id=324.

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

[18] John Field and Carlos A. Varela. Transactors: a programming model for maintaining globally consistent distributed state in unreliable environments. In Jens Palsberg and Martín Abadi, editors, *POPL*, pages 195–208. ACM, 2005.

[19] Cédric Fournet and Georges Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*, pages 372–385, St. Petersburg Beach, Florida, January 21-24 1996. ACM.

[20] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley Professional, 1st edition edition, January 1995.

[21] G. Geerts and William E. McCarthy. An accounting object infrastructure for knowledge-based enterprise models, 1999.

[22] Guido Geerts and William E. McCarthy. The ontological foundation of REA enterprise information systems, August 2000.

[23] Guido Geerts and William E. McCarthy. An ontological analysis of the economic primitives of the extended-REA enterprise information architecture. *International Journal of Accounting Information Systems*, 3(1):1–16, 2002.

[24] Guido L. Geerts and William E. McCarthy. Augmented intensional reasoning in knowledge-based accounting systems. *The Journal of Information Systems*, 2000.

[25] Lukasz Golab and M. Tamer Ozsu. Data stream management issues – a survey. Technical Report CS-2003-08, School of Computer Science, University of Waterloo, April 2003.

[26] Thomas R. Gruber. A translation approach to portable ontology specifications. *Knowledge Acquisition.*, 5(2):199–220, 1993.

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

[28] Kohei Honda and Nobuko Yoshida. A uniform type structure for secure information flow. Technical report, Imperial College London, 2004. Extended abstract appears in Proc. of POPL'2002, Proc. of 29th ACM Symposium on Principles of Programming Languages, ACM Press.

[29] Michael R. A. Huth and Mark Ryan. *Logic in computer science: modelling and reasoning about systems*. Cambridge University Press, 2000.

[30] Mette Jaquet. REAlistic. KSWU thesis, The IT University of Copenhagen, Glentevej 67, 2400 Copenhagen NV, Denmark, March 2003.

[31] O. Jensen and R. Milner. Bigraphs and mobile processes. Technical Report 570, Computer Laboratory, University of Cambridge, 2003.

[32] Simon Peyton Jones, Jean-Marc Eber, and Julian Seward. Composing contracts: an adventure in financial engineering (functional pearl). In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 280–292. ACM Press, 2000.

[33] Nickolas Kavantzas, David Burdett, Gregory Ritzinger, Tony Fletcher, and Yves Lafon. Web services choreography description language version 1.0 – w3c working draft 17 december 2004. Technical report, OASIS Open, Inc., December 2004.

[34] Bartosz Kiepuszewski. *Expressiveness and Suitability of Languages for Control Flow Modelling in Workflows*. PhD thesis, Queensland University of Technology, 2002.

[35] E. Kindler. On the semantics of EPCs: A framework for resolving the vicious circle. Technical report, Reihe Informatik, University of Paderborn, Paderborn, Germany, August 2003.

[36] Naoki Kobayashi. Type systems for concurrent programs. In Bernhard K. Aichernig and T. S. E. Maibaum, editors, *10th Anniversary Colloquium of UNU/IIST*, volume 2757 of *Lecture Notes in Computer Science*, pages 439–453. Springer, 2002.

[37] Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. Linearity and the Pi-Calculus. *ACM Transactions on Programming Languages and Systems*, 21(5):914–947, 1999.

[38] James C. Lampe. Discussion of an ontological analysis of the economic primitives of the extended-REA enterprise information architecture. *International Journal of Accounting Information Systems*, 3(1):17–34, 2002.

[39] David C. Luckham and Brian Frasca. Complex event processing in distributed systems. Technical report, Computer Systems Lab, Stanford University, August 1998.

[40] Henning Makholm and Joe Wells. Instant polymorphic type systems for mobile process calculi: Just add reduction rules and close. In *Proceedings of ESOP 2005*, 2005.

[41] William E. McCarthy. The REA accounting model: A generalized framework for accounting systems in a shared data environment. *The Accounting Review*, LVII(3):554–578, July 1982.

[42] L. G. Meredith and Steve Bjorg. Contracts and types. *Commun. ACM*, 46(10):41–47, 2003.

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

[44] Robin Milner. *Communicating and mobile systems: the π-calculus*. Cambridge University Press, 1999.

[45] W.M.P. van der Aalst Moe Thandar Wynn, David Edmond and A.H.M. ter Hofstede. Achieving a general, formal and decidable approach to the OR-join in workflow using reset nets. Technical Report FIT-TR-2004-02, Queensland University of Technology, Brisbane, 2004. QUT Technical Report.

[46] Arturo Azcorra Salo na, Juan Quemada Vives, and Santiago Pavón Gómez. An introduction to lotos. Technical report, Universidad Politécnica de Madrid, May 1993.

[47] Michael Nissen. Verification of temporal properties for contract specifications, b.sc. thesis. Master's thesis, Department of Computer Science, University of Copenhagen (DIKU), 2005.

[48] James Odell. Power types. *JOOP*, 7(2):8–12, 1994.

[49] C. Ouyang, W.M.P. van der Aalst, S. Breutel, M. Dumas, A.H.M. ter Hofstede, and H.M.W. Verbeek. Formal semantics and analysis of control flow in ws-bpel. Technical Report BPM-05-13, BPM Center Report, 2005.

[50] Robert Paige and Shaye Koenig. Finite differencing of computable expressions. *ACM Transactions on Programming Languages and Systems*, 4(3):402–454, July 1982.

[51] *High-level Petri Nets – Concepts, Definitions and Graphical Notation*, final draft international standard iso/iec 15909, version 4.7.3 edition, May 2002.

[52] 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.

[53] Frank Puhlmann and Mathias Weske. Using the pi-calculus for formalizing workflow patterns. In *Business Process Management, Third International Conference, BPM 2005, Nancy, France, September, 2004. Proceedings.*, LNCS. Springer Verlag, 2005.

[54] Sriram K. Rajamani and Jakob Rehof. Conformance checking for models of asynchronous message passing software. In Ed Brinksma and Kim Guldstrand Larsen, editors, *CAV*, volume 2404 of *Lecture Notes in Computer Science*, pages 166–179. Springer, 2002.

[55] Peter Rittgen. Quo vadis EPK in ARIS ? - Ansätze zu syntaktischen Erweiterungen und einer formalen Semantik. *Wirtschaftsinformatik*, Heft 1:27–35, February 2000.

[56] N. Russell, A.H.M. ter Hofstede, D. Edmond, and W.M.P. van der Aalst. Workflow data patterns. Technical Report FIT-TR-2004-01, Queensland University of Technology, Brisbane, 2004.

[57] N. Russell, A.H.M. ter Hofstede, D. Edmond, and W.M.P. van der Aalst. Workflow resource patterns. BETA Working Paper Series WP 127, Eindhoven University of Technology, Eindhoven, 2004.

[58] Davide Sangiorgi and David Walker. *The π-calculus: A Theory of Mobile Processes*. Cambridge University Press, 2001.

[59] August-Wilhelm Scheer. *Architektur integrierter Informationssysteme: Grundlagen der Unternehmensmodellierung*. Springer-Verlag, 1992.

[60] Alec Sharp and Patrick McDermott. *Workflow Modeling - Tools for Process Improvement and Application Development*. Artech House Publishers, 2000.

[61] John F. Sowa. *Knowledge Representation: Logical, Philosophical, and Computational Foundations*. Brooks Cole Publishing Co., 2000. ISBN 0-534-94965-7.

[62] Christian Stefansen. SMAWL: A SMAll Workflow Language based on CCS. In *Proceedings of the CAiSE Forum of the 17h Conference on Advanced Information Systems Engineering*, June 2005.

[63] 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.

[64] Sun Microsystems. *Solaris 10 Software Developer Collection Multithreaded Programming Guide*.

[65] Satish Thatte, Assaf Arkin, Sid Askary, Francisco Curbera, Yaron Goland, Neelakantan Kartha, Canyang Kevin Liu, Prasad Yendluri, and Alex Yiu. Web services business process execution language version 2.0 working draft, 13 july 2005. Technical report, OASIS Open, Inc., 2005.

[66] UN/CEFACT. Un/cefact modeling methodology (umm) user guide 6 7. Technical report, CEFACT/TMG/N093, September 2003. V20030922.

[67] Wil M. P. van der Aalst, Lachlan Aldred, Marlon Dumas, and Arthur H. M. ter Hofstede. Design and implementation of the yawl system. In Anne Persson and Janis Stirna, editors, *CAiSE*, volume 3084 of *Lecture Notes in Computer Science*, pages 142–159. Springer, 2004.

[68] Wil M.P. van der Aalst. Workflow patterns.

[69] Wil M.P. van der Aalst and Arthur 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.

[70] Wil M.P. van der Aalst, Arthur H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(3):5–51, 2003.

[71] Wil M.P. van der Aalst, Arthur H.M. ter Hofstede, B. Kiespuszewski, and A.P. Barros. Workflow patterns. Technical report, Eindhoven University of Technology, GPO Box 513, NL-5600 MB Eindhoven, The Netherlands, 2002.

[72] Wil M.P. van der Aalst and Kees van Hee. *Workflow Management: Models, Methods, and Systems*. MIT Press, 2002.

[73] W.M.P. van der Aalst. Pi calculus versus Petri nets: Let us eat "humble pie" rather than further inflate the "Pi hype". http://tmitwww.tm.tue.nl/research/patterns/download/pi-hype.pdf, 2004.

[74] W.M.P. van der Aalst, J. Desel, and E. Kindler. On the semantics of EPCs: A vicious circle. In M. Nüttgens and F.J. Rump, editors, *Proceedings of the EPK*

2002: Business Process Management using EPCs, pages 71–80. Gesellschaft für Informatik, November 2002.

[75] W.M.P. van der Aalst and A.H.M. ter Hofstede. Yawl: Yet another workflow language. *Information Systems*, 30(4):245–275, 2005.

[76] W.M.P. van der Aalst, B.F. van Dongen, J. Herbst, L. Maruster, G.Schimm, and A.J.M.M. Weijters. Workflow mining: a survey of issues and approaches. *Data and Knowledge Engineering*, 47(2):237–267, 2003.

[77] Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall. A note on distributed computing. Technical Report TR-94-29, Sun Microsystems, November 1994.

[78] Wikipedia, the free encyclopedia. The Wikimedia Foundation.

[79] Petia Wohed, Wil M.P. van der Aalst, Marlon Dumas, Arthur H.M. ter Hofstede, and Nick Russell. Pattern-based analysis of uml activity diagrams. http://is.tm.tue.nl/research/patterns/download/uml2patterns/20BETA%20TR.pdf, 2004.

[80] The Zing model checker. http://research.microsoft.com/zing/.

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