POETS: Process-Oriented Event-driven Transaction Systems ⋆

Fritz Henglein, Ken Friis Larsen, Jakob Grue Simonsen, Christian Stefansen

Department of Computer Science, University of Copenhagen (DIKU)

Abstract

We present a high-level enterprise system architecture that closely models the domain ontology of resource and information flows in enterprises. It is:

Process-oriented: formal, user-definable specifications for the expected exchange of resources (money, goods, and services), notably contracts, are represented explicitly in the system state to reflect expectations on future events;

Event-driven: events denote relevant information about real-world transactions, specifically the transfer of resources and information between economic agents, to which the system reacts by matching against its portfolio of running processes/contracts in real time;

Declarative: user defined reporting functions can be formulated as declarative functions on the system state, including the representations of residual contractual obligations.

We introduce the architecture and demonstrate how analyses of the standard reporting requirements for companies—the income statement and the balance sheet—can be used to drive the design of events that need registering for such reporting purposes. We then illustrate how the multi-party obligations in trade contracts (sale, purchase), including pricing and VAT payments, can be represented as formal contract expressions that can be subjected to analysis.

To the best of our knowledge this is the first architecture for enterprise resource accounting that demonstrably maps high-level process and information requirements directly to executable specifications.

Key words: process, contract, type, event, financial system, enterprise resource planning, ERP

1 Introduction

Enterprise Resource Planning (ERP) systems integrate several information systems of an organization into one system. Financials, manufacturing, project management, supply chain management, human resource management, and customer relationship management are typical components of an ERP system. [1]

ERP systems do in principle a simple thing: they model activities in an enterprise and register relevant information about them so they can be queried and interpreted for whatever is deemed important or required to run the company, ranging from high-level strategy to down-on-the-floor operations.

It may be surprising then to learn that even ERP systems targeted at small and medium-sized enterprises such as Microsoft Dynamics NAV [2] or AX [3] comprise several million lines of code and thousands of tables in a relational database management system. The software architecture of such systems typically consists of a decomposition of the system in terms of tables, code units, (user interface) forms specifications, data mappers for input/output, etc., built as a three-tier client-server system running on centralized database servers for their data repository. They do not reflect the "architecture" of enterprises, which consists of resources (goods, services, money) being bought, processed, sold, and moved around between different parts, whether physical, functional or organizational, of a company. As a result the translation of business requirements into running code has to span a large and costly semantic and architectural divide.

Despite the widespread use and business reliance on ERP systems little effort has been spent applying sound theoretical principles to designing an ERP system from first principles. We argue that using well-known principles from process algebra and functional programming, as we do here, gives an elegant architecture for ERP systems that more directly reflect business processes and business intelligence needs.

⋆ This work has been supported by the Danish National Advanced Technology Foundation under Project *3rd generation Enterprise Resource Planning systems (3gERP)*. See `http://www.3gERP.org`.

Email addresses: `henglein@diku.dk` (Fritz Henglein), `kflarsen@diku.dk` (Ken Friis Larsen), `simonsen@diku.dk` (Jakob Grue Simonsen), `cstef@diku.dk` (Christian Stefansen).

[1] `http://en.wikipedia.org/wiki/Enterprise_resource_planning` retrieved on June 10th 2008.

[2] `http://www.microsoft.com/dynamics/nav` retrieved on June 10th 2008.

[3] `http://www.microsoft.com/dynamics/ax` retrieved on June 10th 2008.

Taking as our fundamental goal that the *ontological* architecture for requirements also be the architecture, [4] we develop an event-driven architecture aimed at directly reflecting the domain-oriented requirements. The key motivation is shortening the distance between requirements and their formal expression for rapid system prototyping, implementation, and continuous system adaptation to changing processes and information needs.

Given the size of ERP systems, it is not possible to cover all functionality. Hence this paper restricts itself to only consider some of the functionality typically contained in the finance module of ERP systems.

1.1 Contributions

The paper contributes the following:

An ERP system model An ERP system model that
- directly and declaratively reflects *enterprise domain concepts*, notably resources, events, agents, report functions and processes
- does not encumber the system with non-enterprise concepts such as "database management system", "client-server", memory management, etc.
- separates interpretation and registration of (business) events
- separates (monetary) valuation from resources and thus enables re-valuation
- supports user-defined contract specifications, which can be executed and analyzed

Design methodology An enterprise design methodology based on identifying relevant events to be registered and, consequently, processes to be modeled from reporting requirements

Formal semantics A formal semantics (architecture reduction semantics) that is
- event-trace based, yet
- orthogonal to the contract (choreography) language, rendering the concrete choice of language independent of the architecture. [5]

Prototype Illustrative parts of a prototype with a text-based interface, implemented in *F#*.

[4] Motto: the *formalized* requirements *are* the system.

[5] We have used the contract language by Andersen *et al.* [AEH+06]. Other languages such as WS-CDL are conceivable.

In Section 2 we address the question of what to model, i.e., represent as data, and what not. We start with the premise that only the information required for reporting purposes need be modeled. We take the standard reports that every company must supply as our starting point. Section 3 formalizes the entities discovered in this process: agents, resources, valuations of resources (prices), and events that must me registered for the given reporting requirements. Section 4 illustrates how declarative reports can be mapped in a natural fashion to the functional programming language F#.

Section 5 discusses the need for representing not only *ex post* events, but also processes, specifically *contracts*. Section 5.2 describes the resulting architecture: incoming events modeling real-world activities are matched against process/contract states expressing current expected/legal events and then put into a log. In Section 5.3 we describe an example of a contract specification language and show how it integrates with the overall architecture.

We discuss related work in Section 6 and conclude in Section 7.

2 Domain model of resource accounting

In this section we derive a stringent description of the functionality of any system that models the *economic status* of a company. Initially, we notice that at any given point in time, such a status can be derived if we have registered with sufficient granularity the *events* that have occurred up to that point. Events are any atomic, observable change in the state of the world. The challenge lies in selecting what events to register and what events to ignore—and in particular in doing so without a bias to existing methods of accounting.

Receiving an amount into the company's bank account seems inherently *relevant*, whereas the acquisition of a cup of coffee from the machine on the second floor by the clerk may keep the coffee-drinking accountant happy for a while, but is unlikely to have a direct, causal, unequivocal, and important effect on the economic status of the company. But this distinction is vague. A better first approximation to the requirements of relevant events for any accounting system is the union of all events that are mentioned in (accounting) legislation and current accounting practice. However, this approach imports exactly the unfortunate bias towards existing accounting practices which we seek to avoid. For example, many simple accounting systems do not register when a customer accepts a quote and it becomes an order. This is because

such an event has no *direct* effect on any account or in a traditional ledger or on the income statement. Granted, it has the *indirect* effect of starting a process that generally leads to invoicing, and invoicing has a direct effect on an A/R (*Accounts receivable*) account and a revenue account.

In this distinction lies the key to the definition of a *relevant* event: an event is relevant for a particular set of reports if (the result of) at least one of the reports depends on it. Relevance is thus not an intrinsic property of an event, but of what information is dependent on it. To wit, for standard financial reports *by themselves* orders are irrelevant, but for production planning they most certainly are relevant. This leads to a pleasant and quite obvious definition of relevant: an event is relevant if and only if it has a direct effect on any of the reports of the company status that we want the system to produce.

The first step is to determine what reports will be needed. Once these are established, we can proceed to find the event types that affect those reports.

2.1 Reports

We assume that the company needs to produce the following five reports: an income statement, a balance sheet, a cash flow statement, a list of open (not yet paid) invoices, and a VAT (value-added tax, somewhat similar to sales tax) report. These are chosen because they constitute the core functionality of the traditional accounting system that is our benchmark—as well as the legal requirements faced by any registered company. For brevity of exposition, however, we shall concern ourselves with only two reports: the income statement (Figure 1) and the balance sheet (Figure 2).

We now consider as an example a company that sells goods. That is, the company sustains itself by buying goods and selling them with a profit.

For ease of exposition we shall ignore taxes (other than VAT), interests, mortgages, and other advanced accounting phenomena. We argue that this is without loss of generality, as these are similar to the basic accounting phenomena outlined here.

What follows are bare-bones definitions of the reports. For a more thorough exposition the reader is referred to a standard text on accounting, e.g., Weygandt *et al.*'s book [WKK04].

	Revenue (Gross income)
–	Cost of goods sold
=	**Contribution margin**
–	Fixed costs
–	Depreciation
=	**Net operating income**

Fig. 1. **Income Statement** The Income Statement summarizes the profits and losses of the company over a given period (hence also the name *Profit & Loss Statement*).

Assets	*Liabilities and owners' equity*
Fixed assets	**Liabilities**
Current assets	Accounts payable
Inventory	VAT payable
Accounts receivable	**Owners' equity**
Cash and cash equivalents	*Total liabilities and owners' equity*
Total assets	

Fig. 2. **The Balance Sheet** The Balance Sheet summarizes assets, liabilities, and owners' equity at a particular point in time. The balance sheet should always satisfy the fundamental invariant known as the *Accounting Equation*, which states that $Assets = Liabilities + Owners' \ equity$.

2.2 Events

To be able to generate the reports outlined above we must identify the changes in the state of the world that affect each report. Such changes in the state of the world are reported as *events*, and we will assume, based on the *Theory of the Firm* [Cre75], that the events of interest are transfers of economic resources or information between self-interested agents (economic entities).

All that happens can be expressed in terms of a few basic types of events:

- Transmit a resource or money from one agent to another
- Convey information from one agent to another
- Transform a set of resources into another set of resources

These events can (and should) also be further refined, which is what we will do next. Receiving a resource can be something for the company (land, property, paper clips) or something intended for selling with a profit. Some resources are put in stock for later consumption or sale, whereas other resources are consumed the moment they are received (e.g., a session with a business consultant).

Events that affect the Income Statement

Revenue Affected by sending an invoice for normal sale (not fixed assets, for instance) to a customer.

Cost of goods sold Affected by making an inventory requisition relating to a customer order. Notice that the requisition event does not inherently contain information about the purchase price, and thus the purchase price must be looked up or computed. The time of registration varies, but commonly the cost of goods sold is registered at the time where the sale is invoiced to the customer.

Fixed costs Affected by receiving an invoice for a fixed cost.

Depreciation Not an event, but a continuous process. Here depreciation is *computed* (based on the events describing purchases and sales of assets). That is, depreciation is *computed* as a report: it is a function of the registered *bona-fide* "real-world" events. If depreciation is registered as discrete "phantom" events, as present accounting practice mandates, it is difficult to change the depreciation method retro-actively, add a new one, used multiple methods simultaneously, *etc.* In contrast, in our approach real-world events and accounting actions are strictly separated.

Events that affect the Balance Sheet

Fixed assets Affected (a) by receiving an invoice for a fixed asset or (b) by sending an invoice for a fixed asset

Raw materials Affected (a) by receiving an invoice for raw materials or (b) by making a requisition for raw materials from the inventory for production.

Finished goods Affected by (a) sending an invoice to a customer for a good or (b) by receiving finished goods from the production process.

Accounts receivable Affected by (a) sending an invoice or credit note to anyone or (b) receiving payment pertaining to an invoice into the cash register or the bank account

Cash Affected by (a) money being put in the cash register or (b) money being

1	Receive 3 iPhones and 2 MacBooks from supplier X
2	Receive 3 iPhones and 2 MacBooks from supplier Y
3	Receive an invoice from X for 3 iPhones (3 * 2000 DKK incl. VAT) and 2 MacBooks (2 * 10000 DKK incl. VAT) and rush delivery charge (100 DKK, VAT exempt)
4	Receive invoice from Y for 3 iPhones (3 * 2100 DKK incl. VAT) and 2 MacBooks (2 * 9700 DKK incl. VAT) and shipping (500 DKK incl. VAT)
5	Deposit 26100 DKK into X's bank account
6	Send check to Y to the amount of 26200 DKK
7	Observe on our bank account that the check has been cashed
8	Receive order from A of 1 MacBook and 1 iPhone priced at a 15000 DKK incl. VAT for both (bundled)
9	Deliver and invoice 1 MacBook and 1 iPhone to A as ordered
10	Receive from A 15000 DKK into our bank account
11	Settle VAT: Receive 7440 DKK from the VAT tax authority
12	Deliver, invoice, and receive payment for 1 MacBook worth 20000 DKK incl. VAT to Z

Fig. 3. An example of events relevant to a company

taken from the cash register

Bank account Affected by (a) money being deposited into our bank account or (b) money being withdrawn from out bank account

Accounts payable Affected by (a) receiving an invoice or credit note from anyone or (b) sending payment pertaining to an invoice from the cash register or the bank account

VAT payable Affected by issuing or receiving an invoice containing items on which VAT is due.

Owners' equity Affected by transferring money or a resource to and from the owners.

Figure 3 shows an example of the events that need to be registered by a company over a period of time.

3 Formal model of resource accounting

3.1 Agents

Agents represent whole companies as well as categorizations within a company such as organizational unit, location, etc. They can be thought of as partitioning a company, possibly along multiple dimensions for any suitable purpose. To be able to distinguish resources determined for (re)sale, for use inside the company, but with long-term depreciation or instantaneous depreciation, we require conceptual *resource containers* such as *operations*, *fixed assets*, *losses*. In our model agents are thus not restricted to modeling only legal persons, organizational units, roles, or actual persons in the real world, as in the REA-model [McC82].

We have seen that for the reporting purposes presented and analyzed in Section 2 it is sufficient to have an agent representing a company and a set of internal agents, where each internal agent has a unique company that it belongs to. This can be captured by defining

$$\begin{aligned} \mathit{Agent} &= \mathit{CompanyName} \times \mathit{InternalAgentName} \\ \mathit{CompanyName} &= \mathit{String} \\ \mathit{InternalAgentName} &= \mathit{String} \end{aligned}$$

where the empty string, ϵ, also written as "me", is the designated internal name for the company itself. We write $C.I$ instead of (C, I) and C instead of $C.\epsilon$. We write $A \leq C$ if $A = (C, I)$ for some $I \in \mathit{InternalAgentName}$.

3.2 Resources

A *resource* is either: empty; a unit of *resource type* identified by a unique resource name such as *(one) iPhone*, *(1 liter of) water*, Picasso's *Guernica* painting; a scaled resource, e.g. 2 *iPhone*; or the formal sum of two resources, which models taking their union, e.g. $2\,\mathit{iPhone} + 1.4\,\mathit{water}$. Note that "a set of resources" is "a" resource—in the singular. If a resource consists of more than one particular resource type, such as the iPhones and the water in the last example, we call it a *compound* resource. Note the use of the singular "a" if it is intuitively plural ("a set of resources").

Resource types are usually categorized into unique and nonunique, meaning Picasso' Guernica is a unique resource, water and iPhones are normally nonunique treated as nonunique resources. Furthermore, nonunique resources

can be scaled discretely or continuously: there can only be an integral number of iPhones, but water may be managed in arbitrary fractions of liters. These distinctions translate into how resource types can be scaled, corresponding to allowing $\{0,1\}$, $\mathbb{N}$ or $\mathbb{R}_0^+$ as scaling factors. We shall not distinguish between resource types here, but model all resource types as being continuous. To simplify the presentation of the semantic resource model we shall not dwell on the handling of unique and discrete resources, but treat all resources as continuously scalable. This is without loss of expressive power since $\mathbb{N}$ and $\{0,1\}$ can be embedded into the nonnegative reals, and we can maintain a mapping from resource types to their category and referring to it during computations on resources to ensure that the corresponding invariant is satisfied. Finally, we also allow "negative" resources, which ensures that the difference between resources is always defined.

Stipulating the existence of a countably infinite set *ResourceName* of different resource types, each identified by a unique string, the smallest set of resources closed under the constructions above is the set $\mathit{ResourceName} \hookrightarrow \mathbb{R}$ of finite partial maps from *Resource* to $\mathbb{R}$.

It is worthwhile observing that $\mathit{ResourceName} \hookrightarrow \mathbb{R}$ together with scaling and addition satisfies the axioms of a *vector space*. Since the particular names chosen for resource types are irrelevant we might as well identify them with the natural numbers. This amounts to resources being isomorphic with the *infinite coordinate space* $\mathbb{R}^\infty$ over the field $\mathbb{R}$. This is an infinite dimensional vector space whose elements are infinite vectors of reals $(k_1, k_2, ...)$ or, equivalently, formal sums $\sum_{i=1}^{\infty} k_i X_i$ with finitely many nonzero elements k_i. Understood as a resource such a vector or formal sum indicates in its k_i-th component how many units of resource type i (respectively X_i) are part of it.

For convenience we shall continue using the descriptive strings as resource names instead of formals X_i when giving examples: $2\,\mathit{iPhone} + 3\,\mathit{MacBook}$ denotes the compound resource consisting of 2.0 times one unit of the resource type denoted by *iPhone* plus 3.0 times one unit of the resource type denoted by *MacBook*. This is instead of writing $2X_{234}+3X_{4117}$ or $(0,\ldots,0,2,0,\ldots,0,3,0,\ldots)$ where the 2 occurs in the 234th component and the 3 in the 4117th component of the infinite vector and where 234 is the *item number* of iPhones and 4117 is the item number of MacBooks.

To summarize, we have the following spaces for modeling resource names and resources

$$\begin{aligned} \mathit{ResourceName} &= \mathit{String} \\ \mathit{Resource} &= \mathit{ResourceName} \hookrightarrow \mathbb{R} \qquad (\cong \mathbb{R}^\infty) \end{aligned}$$

Note that the isomorphism between vector spaces $\mathit{ResourceName} \hookrightarrow \mathbb{R}$ and

$\mathbb{R}^\infty$ corresponds to a *product catalog*, which maps between descriptions of resource types and their items numbers.

We need an additional vector operation in connection with costing, which we introduce now. We say resource $R = \sum_{i=0}^{\infty} k_i X_i$ is *nonnegative* and write $R \geq 0$ if $k_i \geq 0$ for all $i \in \mathbb{N}$. We write $R \leq R'$ if $R' - R \geq 0$.

We define the operation $\mathit{Subtract} : \mathit{Resource} \times \mathit{Resource} \to \mathit{Resource} \times \mathit{Resource}$ as follows: for $R_1, R_2 \geq 0$ we define $\mathit{Subtract}(R_1, R_2) = (R_1', R_2')$ if (1) $R_1', R_2' \geq 0$; (2) $R_1 + R_2' = R_1' + R_2$, and (3) R_1', R_2' are least with respect to $\leq$ amongst vectors that have the first two properties.

Subtract models the situation where we would like to subtract a compound resource R_2 from another resource R_1. If there are more units of a particular resource type in R_1 than in R_2, after subtracting those in R_2 from those in R_1 the remaining units are returned as part of R_1', with R_2' receiving 0. If there are fewer, the roles of R_1' and R_2' are reversed. E.g.,

$$\mathit{Subtract}(2\,\text{iPhone} + 1\,\text{MacBook} + 1\,\text{Guernica}, 1\,\text{iPhone} + 5\,\text{MacBook}) = (1\,\text{iPhone} + 1\,\text{Guernica}, 4\,\text{MacBook}).$$

3.3 Valuations

We define a *valuation* to be a map from *Resource* to a subspace *ValResource* of *Resource*. A valuation expresses what general resources are *worth* in terms of designated (other) resource types. The subspace *ValResource* can, in principle, be arbitrary, but is normally spanned by *currencies* such as USD, EUR, JPY, DKK. In other words, a valuation normally maps goods and services to money. In practice valuations map to a single currency, which requires mapping also money in other currencies. In this case *ValResource* is isomorphic to $\mathbb{R}$, the amounts in some designated currency, say, DKK. We shall take this as our default case below, but remark that the ensuing presentation generalizes to arbitrary subspaces of *Resource*.

Valuations are used for a number of purposes: The most obvious is as a *price list* as specified in the prices posted in a shop or, indirectly, as part of the line items of a sales contract. Another is to express the result of manual (re)appraisal of certain resources a company may have, e.g., in connection with extraordinary write-downs of assets.

Valuations distribute over scaling of resources and taking their union (sum):

$$\mathit{Value}(R_1 + R_2) = \mathit{Value}(R_1) + \mathit{Value}(R_2)$$

$$Value(kR_1) = k\,Value(R_1)$$

In other words, the total value of two resources R_1 and R_2 is the sum of their values; and the value of k copies of a resource is k times the individual cost. This is tantamount to saying that valuations are *linear functions (homomorphisms)* between the vector spaces *Resource* and $\mathbb{R}$. In particular, a valuation is by given canonically by providing the value of one unit of each resource type.

$$Valuation = Hom(Resource, ValResource) \qquad (\text{Here } ValResource \cong \mathbb{R})$$

The linearity requirement does not force pricing in sales contracts to apply the same unit price in every trade. Larger volumes may just let the buyer negotiate a lower price, corresponding to a *different* valuation as the basis for a sales contract. Linearity rather models *distributing* value of resources when they are split up as part of movement to and from inventory, production units, *etc.* Here the basic assumption is always that items acquired in the same transaction are all equally priced. So, if 8 iPhones are bought for stock and 4 of them are transferred to operations later on, the original 4 and the transferred ones are given the same unit price, whatever that is.

3.4 Events

As stated in Section 2.2 we need events to model transmission of resources, transformation of resources, and conveyance of information. We will refer to these collectively as *transactional events*—or in the definition *TransactEvent*. Each event is equipped with a *time stamp* and an *identifier* for correlating different events to each other. Timestamps are represented by real numbers that model the time difference to some given base point (say, January 1, 1970, 00:00:00). We use strings for identifiers.

We arrive at the following definitions:

$$\begin{aligned} Event &= LogEvent \times (Time \times Ident) \\ LogEvent &= TransactEvent \end{aligned}$$

$$\begin{aligned} TransactEvent &= TransmitEvent \uplus TransformEvent \uplus InformEvent \\ Time &= \mathbb{R} \\ Ident &= String \\ TransmitEvent &= Agent \times Agent \times Resource \\ TransformEvent &= Agent \times Resource \times Resource \end{aligned}$$

$$InformEvent = Agent \times Agent \times Information$$

Note that $\uplus$ denotes disjoint union.

We are left with modeling the information conveyed in information events. The only information events that need be captured for the given reporting purposes is the transmission of *invoices*. An invoice carries a lot of information in practice: sender and receiver, their contact information, company tax code information (for VAT purposes), which resources are delivered, and expenses for delivering them such as shipping and handling. The resources are usually also split up into resource names, number of units (scale), and what each unit costs.

Sender, receiver, time, and an identification are conveyed as part of the event itself. The remaining information needed we represent in the information part. Note that we only capture information that is required for our reporting purposes! This consists of the resources delivered/to be delivered and price information:

$$Information = Resource \times PriceInformation$$

The price information, in turn, consists of a *price* of the resources and their *value added tax (VAT)*.

$$PriceInformation = Price \times VAT$$

The price is a valuation. To simplify matters we disregard the rather complex conglomerate of legislation on VAT calculation, and represent VAT as a valuation as well. It is possible to factor VAT valuations into a VAT *rate* and its application to the resource price. Since VAT rate and VAT valuation are intercomputable given the price, we shall freely use one or the other below.

$$Price = Valuation$$
$$VAT = Valuation$$

Note that an element $(r, (p, t)) \in Information$ is a 3-tuple, where each component is essentially a map whose domain is contained in $ResourceName$. The price, p, and VAT, t, components have an infinite domain, but only those resource types in the (finite) domain of r are relevant. So in essence the three

components each represent a separate map from a finite set of resource types: For r the mapping goes to $\mathbb{R}$, the number of units of each resource type; for p and r it goes to *ValResource*, the price, resp. VAT per unit resource. The three maps can be combined to a single one, of course, which is exactly how invoices are formulated in practice: A *line item* contains the item (resource type), the number of units delivered, and price and VAT per unit or for all units, or some equivalent formulation containing the same information.

Note that we can model bundled prices. A bundle is a compound resource, but with its own price different from the valuation of the bundle in terms of its constituent resources. This can be modeled by introducing a bundle as an "abstract" new resource resource type, whose "implementation" (definition) is the compound resource it is made up of. This allows pricing of that resource independently of its constituent resources. [6]

Beyond the above information an invoice also contains payment terms and other information. Following our principle of capturing information that is required to produce our target reports and no more, payment terms are not registered, however. In practice, a reference to the actual invoice from the information event will give access to all such information.

With the above formalization the events of the example in Figure 3 can be rendered as shown in Figure 4. A line item consists of resource type, units delivered, price per unit and VAT rate. Notice that the registered events are a transcription of the corresponding real-world events, without interceding bookkeeping artifacts.

3.5 Reports

Let us reconsider our reports from Section 2. They are parameterized over

- a time period, with starting date and end date, and
- a time-stamped set of events.

Note that the first part also holds for the balance sheet: the difference to the income statement and other *periodic* statements is that the period of interest for it usually covers the starting date of the company until a particular period end date, whereas the other reports use a more recent start date.

The time period is solely used to filter out events outside the designated period. Once that is done, each report can then be defined as a function

[6] This is tantamount to treating a bundle as a *product* whose bill of materials is the compound resource it is made up of.

1	Receive 3 iPhones and 2 MacBooks from supplier X	transmit(X, Me, 3 iPhone + 2 MacBook, 2008-01-15)
2	Receive 3 iPhones and 2 MacBooks from supplier Y	transmit(Y, Me, 3 iPhone + 2 MacBook, 2008-01-19)
3	Receive an invoice from X for 3 iPhones (3 * 2000 DKK incl. VAT) and 2 MacBooks (2 * 10000 DKK incl. VAT) and rush delivery charge (100 DKK, VAT exempt)	inform(X, Me, items {(iPhone, 3, 1600 DKK, 25%), (MacBook, 2, 8000 DKK, 25%), (fee, 1, 100 DKK, 0%)}, 2008-01-19)
4	Receive invoice from Y for 3 iPhones (3 * 2100 DKK incl. VAT) and 2 MacBooks (2 * 9700 DKK incl. VAT) and shipping (500 DKK incl. VAT)	inform(Y, Me, items {(iPhone, 3, 1680 DKK, 25%), (MacBook, 2, 7760 DKK, 25%), (shipping, 1, 400 DKK, 25%)}, 2008-01-19)
5	Deposit 26100 DKK into X's bank account	transmit(Me.bank, X.bank, 26100 DKK, 2008-01-22)
6	Send check to Y to the amount of 26200 DKK	(not a relevant event for standard financial reports)
7	Observe on our bank account that the check has been cashed	transmit(Me.bank, Y.bank, 26200 DKK, 2008-01-24)
8	Receive order from A of 1 MacBook and 1 iPhone priced at 15000 DKK incl. VAT	enter sales order (contract) with items { (bundle1, 1, 12000 DKK, 25%)} where bundle1 = 1 iPhone + 1 MacBook (see Section 5)
9	Deliver and invoice 1 MacBook and 1 iPhone to A as ordered	transmit (Me, A, 1 bundle1, 2008-01-26); inform(Me, A, items {(bundle1, 1, 12000 DKK, 25%)}, 2008-01-26)
10	Receive from A 15000 DKK into our bank account	transmit (A.bank, Me.bank, 15000 DKK, 2008-01-30)
11	Settle VAT: Receive 7440 DKK from the VAT tax authority	transmit (Skat, Me.bank, 7440 DKK, 2008-02-08)
12	Deliver, invoice, and receive payment for 1 MacBook worth 20000 DKK incl. VAT to Z	transmit (Me.ops, Z, 1 MacBook, 2009-01-30); inform (Me, Z, items {(MacBook, 1, 16000 DKK, 25%)}, 2009-01-30); transmit (Z.bank, Me.bank, 20000 DKK, 2009-02-06);

Fig. 4. Example with formal events

$$
\begin{aligned}
\mathit{InvoicesReceived} &= \{(i,(A,R,(\mathit{priceInfo},t))) \\
&\quad : (\mathit{inform}(A,B,(R,\mathit{priceInfo})),t,i) \in \mathit{Events} \mid B \leq \mathit{me}, A \not\leq \mathit{me}\} \\
\mathit{InvoicesSent} &= \{(i,(B,R,(\mathit{priceInfo},t))) \\
&\quad : (\mathit{inform}(A,B,(R,\mathit{priceInfo})),t,i) \in \mathit{Events} \mid A \leq \mathit{me}, B \not\leq \mathit{me}\}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{InvAcq} = \mathit{Sort}\{&(R,(\mathit{priceInfo},t)) \\
&: (\mathit{transmit}(\mathit{me},\mathit{me}.\mathit{Inventory},R),t,i) \in \mathit{Events}, \\
&\quad (A,R',\mathit{priceInfo}) = \mathit{InvoicesReceived}(i)\}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{GoodsSold} = \sum\{R : (i,(A,R,(\mathit{priceInfo},t))) \in \mathit{InvoicesSent}\} \\
&\mathit{FIFOCost} = \mathit{foldl}(\mathit{accumCost},(0,\mathit{GoodsSold}),\mathit{InvAcq})
\end{aligned}
$$

where

$$
\mathit{foldl}(f,e,[x_1,x_2,...,x_n]) = f(x_n,\ldots f(x_2,f(x_1,e))\ldots)
$$

$$
\begin{aligned}
&\mathit{accumCost}((R,((p,m),t)),(\mathit{total},Q)) = \\
&\quad \text{let } (R',Q') = \mathit{Subtract}(R,Q) \text{ in} \\
&\qquad (\mathit{total} + p(R-R'),Q') \\
&\quad \text{end}
\end{aligned}
$$

Fig. 5. Definitions of selected subreports

on the remaining events.

In other words, each report is the result of composing a filtering function with a function that processes the resulting time-stamped events. It is the latter function we henceforth treat as the reporting function *proper* when talking about income statement, balance sheet, and the other reports.

3.5.1 Subreports

In this section we consider some subreports (auxiliary functions) that are necessary to create the income statement and the balance sheet, which are then defined in the next section.

Below we give the definitions of some of the reports. We use set-comprehension notation as it first appeared in the SETL programming language [SDDS86]. These are simplified specifications for reasons of exposition. We assume that all

goods purchased and transferred into inventory are for eventual sale only, and fixed assets are written as if acquired and sold the first day of the accounting period (usually year).

We would like to emphasize that we use the term "report" here to denote any computable function on sets of events. This is in contrast to accounting system practice, where the term usually conveys an expectation that the result be rendered in some graphical format (as a printable document), and where functions computed as part of other systems (such as data warehouses, OLAP engines or spreadsheet applications) may carry other designations than *report* even though they also are definable as computable functions on business events.

Figure 5 contains the mathematical definition of the subreports necessary to define the report *FIFOCost*; all of which will be described shortly. The remaining subreports can be found in Appendix A and Figure A.1.

The *InvoicesSent* and *InvoicesReceived* reports As a basic report we need a map from identifiers to corresponding invoice information. Payment and goods/service delivery events are correlated to invoices via their identifiers that they share with their invoice.

The *InvAcq* report We call a set of resources, where each is associated with a time-stamped price and VAT valuation, *priced resources*.

The *inventory acquisitions* are the priced resources that have been transferred to internal agent *Inventory*, sorted according to their time-stamp. The identifier of an internal transmit event is used to indicate from which original purchase the price information comes.

The *GoodsSold* report The most interesting reports relate to *costing* because they reflect accounting decisions as to attributing a cost (valuation) to goods sold. For unique resources we can uniquely associate a valuation by looking up the price information in the invoice received for it. For nonunique resources, however, many purchases may contain the same resource. Costing is about allocating a valuation to goods sold that is normally derived from their purchase prices. There are several generally accepted methods for inventory valuation: first-in-first-out (FIFO) costing, last-in-first-out (LIFO) costing, average costing, etc. For illustration purposes we use FIFO costing here.

The goods sold in the period are the resources invoiced to another company, which means that they have been or are committed to being delivered. We

	Revenue	$\triangleq \sum\{p(R) : (i, (A, R, ((p, m), t))) \in \mathit{InvoicesSent}\}$
–	Cost of goods sold	$\triangleq \#1(\mathit{FIFOCost})$
=	**Contrib. margin**	$\triangleq \mathit{Revenue} - \mathit{Cost\ of\ goods\ sold}$
–	Fixed costs	$\triangleq \sum\{p(R) : (R, ((p, m), t)) \in \mathit{Expenses}\}$
–	Depreciation	$\triangleq \mathit{Depreciation}$
=	**Net op. income**	$\triangleq \mathit{Contrib.\ margin} - \mathit{Fixed\ costs} - \mathit{Depreciation}$

Fig. 6. **Income Statement** The Income Statement summarizes the profits and losses of the company over a given period (hence also the British term *Profit & Loss Account*).

assume that all such resources must be moved out of inventory in connection with the sale, and that the identifier of the move indicates which sale (invoice) it relates to.

The *FIFOCost* report *FIFOCost* returns the value of goods removed from inventory for sale, combined with any remaining goods that could not be found in inventory. Ordinarily the latter is always 0 for a reporting period. However, the function is general enough to handle cases where items have been sold before they have been moved into inventory.

With the basic reports of the previous section it is possible to define the income statement and the balance sheet. These are shown in Figures 6 and 7.

4 Prototyping in F#

Many details are easy to overlook without a machine-checkable and executable model. For this reason we have produced a proof-of-concept implementation in F# [SGC07], an ML dialect similar to O'Caml. An important point of this section is to show that once we have the rigorous formal model, the actual coding of the system is simple, and even reports that are considered complex in standard ERP systems can be implemented in a succinct way.

We start with some type definitions for modeling agent specifications and resources:

```
type company_name = string
type internal_agent_name = string
type agent_spec = {company: company_name;
```

Assets	
Fixed assets	$\triangleq$ *FAssetAcq* $-$ *Depreciation*
Current assets	$\triangleq$ *Inv.* $+$ *Acc. rec.* $+$ *Cash* $+$ *equiv.*
Inventory	$\triangleq \sum\{p(R) : (R, ((p, m), t)) \in \mathit{InvAcq}\}$
	$-\#1(\mathit{FIFOCost})$
Acc. receivable	$\triangleq \sum\{p(R) : (i, (A, R, ((p, m), t))) \in \mathit{InvoicesSent}\}$
	$-\sum\{a : (i, a) \in \mathit{PaymentsReceived}\}$
Cash+equiv.	$\triangleq \sum\{a : (i, a) \in \mathit{PaymentsReceived}\}$
	$-\sum\{a : (i, a) \in \mathit{PaymentsMade}\}$
Total assets	$\triangleq$ *Fixed assets* $+$ *Current assets*

Liabilities and owners' equity	
Liabilities	
Acc. payable	$\triangleq \sum\{p(R) : (i, (A, R, ((p, m), t))) \in \mathit{InvoicesReceived}\}$
	$-\sum\{a : (i, a) \in \mathit{PaymentsMade}\}$
VAT payable	$\triangleq$ *VATOutgoing* $-$ *VATIncoming*
Owners' eq.	$\triangleq$ *Total assets* $-$ *Liabilities*
Total liab.+eq.	$\triangleq$ *Liabilities* $+$ *Owners' equity*

Fig. 7. **The Balance Sheet** The Balance Sheet summarizes assets, liabilities, and owners' equity at a particular point in time. The balance sheet should always satisfy the fundamental invariant known as the *Accounting Equation*, which states that *Assets* = *Liabilities* + *Owners' equity*.

```
    agent: internal_agent_name option
}

type resource_name = string
type resource = (resource_name * float) list
```

The definitions are straightforward as one would expect. Notice that we naively represent elements of *Resource* for simplicity by association lists.

We model information in invoices as:

```
type invoice_line = {no_items: float;
```

```
                    resource: resource_name;
                    price: int; (* per item in std. currency *)
                    vat: int; (* VAT per item in std. currency *)
                   }
type invoice = invoice_line list
```

This corresponds to the definition of line items as discussed in Section 3. Finally, we model events, log entries, and logs with the following definitions:

```
type ident = string
type log_event =
  | Transmit  of agent_spec * agent_spec * resource
  | Transform of agent_spec * resource * resource
  | Inform    of agent_spec * agent_spec * invoice
type log_entry =
  | Event of log_event * date * ident
type log = log_entry = log_entry list
```

Here we use dates (year–month–day) as timestamps and strings as identifiers.

In the proof-of-concept implementation F# doubles as the preliminary report language, i.e., the implementation of the architecture itself is in F#, and to avoid introducing a separate report language at this stage, F# is also used to write reports. Although F# is quite suitable for that purpose, a complete system would most likely use a report language designed specifically for enterprise reporting rather than a general-purpose language. Reports, for now, are simply F# functions that take the log as an argument (and possibly additional context arguments).

We present two subreports from Section 3.5.1 of varying complexity: one for listing the received invoices, the other for computing the accumulated cost of inventory requisitions using the FIFO method.

Invoices Received The set of invoices received can be found by simple inspection of the log. The following function `invoices_received` runs through the log, finds the relevant `Inform` events, filters out the resources (using the function `choose_informs_where`), and builds an association list for all invoice ids:

```
let choose_informs_where f log =
  let match_trans trans =
    match trans with
    | Event(Inform(s,r,inv), d, id) -> f s r inv d id
    | _ -> None in
  List.choose match_trans log
```

```
let invoices_received me log =
  choose_informs_where (fun sender receiver inv d id ->
    if sender.company = me && receiver.company <> me
    then Some (id, (inv, d))
    else None) log
```

The function `invoices_received` takes two arguments: the name of my company, `me`, and the log. Notice that the function ignores the internal agent specification by only looking at the `company` attribute. Analogously to the utility function `choose_informs_where` that selects information events we define the utility function `choose_transmits_where` for selecting transmit events.

FIFO Inventory Costing To find the cost of what has been taken out of the inventory using FIFO ordering as described in Section 3.5, we define the following function `fifo`:

```
let addf key map f = Map.add key (Map.tryfind key map |> f) map

let fifo inventory time log me =
  let stuff_in_inv =
    choose_transmits_where (fun sender receiver res d id ->
      if receiver = inventory && d <= time then Some(res, d, id)
      else None) log in
  (* Assumes that log is time-sorted, thus inList is also sorted *)
  let inList = List.map (fun (r,t,id) -> lookup_price log me r id)
               stuff_in_inv in
  let price_map =
    List.fold_right (fun inv map ->  (* use fold_left for LIFO *)
      List.fold_left (fun map (name, no, price) ->
        addf name map
          (function
            | Some s -> (no, price) :: s
            | None -> [no,price])) map inv) inList Map.empty in

  let outList =
    choose_transmits_where (fun sender _ res d _ ->
      if sender = inventory && d <= time then Some res
      else None) log in
  (* Outflows *)
  let outSum = List.fold_left add_res null_resource outList in
  let price_atomic (name, total) =
    let prices = Map.find name price_map in
    let rec loop remaining ((n,p)::prices) =
      if remaining > n then (n*p) + loop (remaining - n) prices
      else remaining * p in
    loop total prices in
```

```
  let total_price = List.sumByFloat price_atomic outSum in
  total_price
```

It uses the following helper functions to look up the prices of a resource in the corresponding invoices.

```
let find_in_invoice resource invoice =
  List.map (fun (name, no) ->
    let line = List.find (fun line -> line.resource = name) invoice
    in name, no, line.price) resource
let lookup_price log me r id =
  let match_invoice = function
    | Event(Inform(sender, receiver, lines), _, ident)
      when receiver = me && ident = id -> Some lines
    | _ -> None in
  let invoice = List.first match_invoice log |> Option.get in
  find_in_invoice r invoice
```

5 A contract-oriented event-based architecture

In this section we describe the formal semantics of a contract-oriented event-based architecture. The architecture is deliberately designed to allow any contract language to be used. We begin by providing background, proceed to describe the architecture, and afterwards give a concrete example of a contract language.

5.1 Background

The most basic form of economic interaction is that of an *exchange* of resources between agents. If we take || to be the basic composition operator, we could imagine describing an exchange by writing:

```
transmit (X, Y, 1 apple) || transmit (Y, X, 1 USD)
```

This represents a particular *contract* between X and Y that, if and when it is *entered*, obliges `X` to transfer an apple to `Y` and `Y` to give `X` a dollar in consideration. It says very little else. It sets no time limits, and it mandates no particular ordering. It does, however, say that until an apple has been transmitted and a dollar has been transmitted in consideration hereof, *obligations remain.*

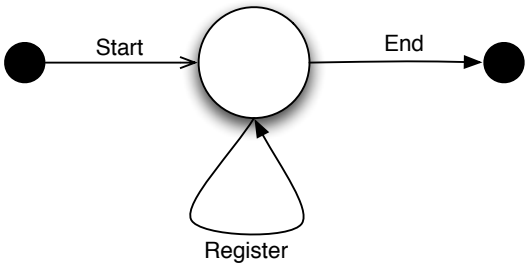

Fig. 8. State diagram showing the life cycle of contracts

If the event occurs that `X` transmits an apple to `Y`, the event should be logged, and, moreover, the state of the contract should now reflect that only one obligation remains. We say that the event *matches* (i.e., satisfies) an obligation in the contract, and the result of matching is a *residual contract* representing the remaining obligations.

In general, it is clear that the state of a company requires representation of the processes that it is committed to following, either for contractual reasons or for non-legal reasons. We suggestively call all such processes contracts, even though they may also represent processes that have no legal significance, such as internal processes.

The portion of the contract *life cycle*, that we need to model is sketched in Figure 8. The steps in the contract life cycle are as follows:

NEGOTIATE The terms are negotiated between two or more parties (independent agents). We do not model this stage.

START The contract is *started* (i.e., entered). At this point the contract describes potentially different series of resource and information transfer steps that must happen over time

REGISTER A transmit or inform event is *matched* against a contract; that is, it is checked to see whether it is a valid step according to the contract. If it is, the event is *registered*, and the contract is updated to represent only the remaining obligations. If it is not, both the offending event and a representation of the residual obligations in the contract at that time are returned for error processing.

END The contract is *ended*. This can happen for a variety of reasons, most commonly because no obligations remain. If a breach of contract has occurred, we might choose to end it, albeit unsuccessfully; what happens thereafter is decided outside of the system.

This means that in addition to logging all transaction events (transmit, transform, and inform), we must log whenever a contract is *started* or *ended*.

All of this leads to an architecture consisting of a contract engine, a log, and a report engine.

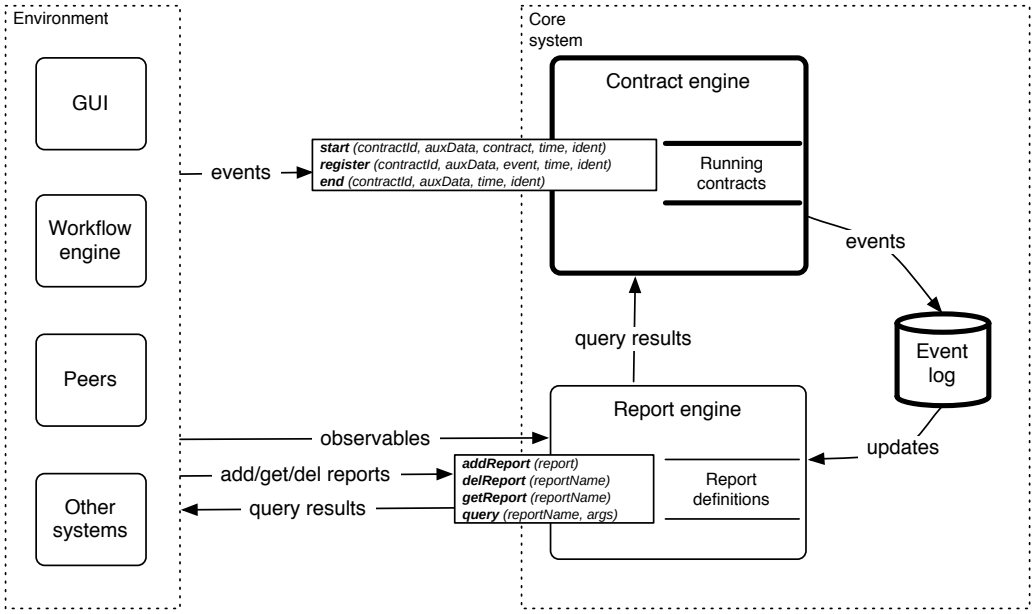

Fig. 9. Event-driven, contract-based architecture. Only the components in bold are treated in this paper; for reports readers are referred to Nissen and Larsen [NL08].

5.2 Architecture

Figure 9 shows a birds-eye view of the architecture. We assume that there is an environment that takes care of collecting and buffering events. These are then matched, manually or automatically, with an ongoing contract. The environment can be a GUI, a workflow engine, or other systems that interact with the contract engine or the report engine.

Since the report engine has been developed in a related paper [LN07], we will concentrate on a formal model of the log and the running contracts. The log, L, is a set containing elements of the type *Event*. The precise structure of the abstract representation, C, of the running contracts depends on the concrete contract language being used. Each of the running contracts can be identified uniquely via a contract identifier, *cid*.

The system changes over time via a sequence of events. Given $e \in \mathit{Event}$, the transition relation on the state of the core system looks as follows

$$\langle L, C \rangle \xrightarrow{e} \langle L', C' \rangle$$

The definition of *Event* from Section 3.4 must be extended (a) to accommodate events that start and end contracts and (b) to allow for any auxiliary data that the contract language may need:

$$
\begin{aligned}
\mathit{Event} &= \mathit{LogEvent} \times (\mathit{Time} \times \mathit{Ident}) \\
\mathit{LogEvent} &= (\mathit{StartEvent} \uplus \mathit{RegisterEvent} \uplus \mathit{EndEvent}) \\
\\
\mathit{StartEvent} &= \texttt{Start} \times \mathit{ContractID} \times \mathit{AuxData} \times \mathit{Contract} \\
\mathit{RegisterEvent} &= \texttt{Register} \times \mathit{ContractID} \times \mathit{AuxData} \times \mathit{TransactEvent} \\
\mathit{EndEvent} &= \texttt{End} \times \mathit{ContractID} \times \mathit{AuxData} \\
\\
\mathit{TransactEvent} &= \mathit{TransmitEvent} \uplus \mathit{TransformEvent} \uplus \mathit{InformEvent} \\
\mathit{Time} &= \mathbb{R} \\
\mathit{Ident} &= \mathit{String} \\
\mathit{TransmitEvent} &= \mathit{Agent} \times \mathit{Agent} \times \mathit{Resource} \\
\mathit{TransformEvent} &= \mathit{Agent} \times \mathit{Resource} \times \mathit{Resource} \\
\mathit{InformEvent} &= \mathit{Agent} \times \mathit{Agent} \times \mathit{Information}
\end{aligned}
$$

A *LogEvent* can now be either a start event, a register event, or an end event. All of these three carry a contract identifier to indicate the contract being started, matched against or ended. They also carry auxiliary data, which are any data specific to the contract language. *StartEvent* additionally contains the body of the contract being inserted into the system, and *RegisterEvent* has a *TransactEvent* as part of its payload. As before all events contain a time stamp of type *Time* as well as an event identifier, *Ident*, chosen by the environment to be able to refer to the event later.

5.2.1 State transitions

We can now begin to consider the state transitions of the core architecture. The transitions are described by the inference rules displayed in Figure 10.

START The START rule inserts a new contract into the system state and logs it. $\mathbf{start}(\mathit{cid}, x, c)@(t, \mathit{id})$ denotes a start event with contract identifier *cid*, auxiliary data x, contract c, time stamp t, and event ID *id*. The contract language-specific operation $C \oplus (\mathit{cid}, x, c)$ adds the contract c with identifier *cid* and auxiliary data x to the running contracts represented by C. The rule applies only if the chosen identifier, *cid*, is, indeed, previously unused.

$$\textsc{start}\ \frac{cid \text{ does not exist in } C \qquad C' = C \oplus (cid, x, c)}{\langle L, C\rangle \xrightarrow{\mathbf{start}(cid,x,c)@(t,id)} \langle (\mathbf{start}(cid, x, c)@(t, id)) :: L, C'\rangle}$$

$$\textsc{register}\ \frac{C \xrightarrow{(cid,x,e)} C'}{\langle L, C\rangle \xrightarrow{\mathbf{reg}(cid,x,e)@(t,id)} \langle (\mathbf{reg}(cid, x, e)@(t, id)) :: L, C'\rangle}$$

$$\textsc{end}\ \frac{C \xrightarrow{(cid,x)} C' \qquad C'' = C' \ominus cid}{\langle L, C\rangle \xrightarrow{\mathbf{end}(cid,x)@(t,id)} \langle (\mathbf{end}(cid, x)@(t, id)) :: L, C''\rangle}$$

Fig. 10. Transition relation for the core architecture

REGISTER In this rule e is the *TransactEvent* payload of the register event. The operational semantics of the REGISTER rule relies on the operational semantics of the contract language: if the contract language permits the transition $C \xrightarrow{(cid,x,e)} C'$, the register event is logged, and C' is the new state of the running contracts.

END The END rule removes a contract, *cid*, from the running contracts, provided that the contract language permits the transition $C \xrightarrow{(cid,x)} C'$. The removal is written as $C \ominus cid$ where $\ominus$ is a contract language-specific operator.

5.3 An example contract language

In this section we show how to describe contracts as compositional specifications in the language of Andersen *et al.* [AEH+06]:

$$\begin{aligned} c ::= {} & \text{Success} \mid \text{Failure} \mid f(\vec{a}) \mid c_1 + c_2 \mid c_1 \parallel c_2 \mid c_1; c_2 \\ \mid {} & \text{transmit}(A_1, A_2, R, T \mid P).\, c \\ \mid {} & \text{transform}(A, R_1, R_2, T \mid P).\, c \\ \mid {} & \text{inform}(A_1, A_2, I, T \mid P).\, c \end{aligned}$$

Success denotes the *trivial* or *(successfully) completed* contract: it carries no obligations on anybody. Failure denotes the *inconsistent* or *failed* contract; it signifies breach of contract or a contract that is impossible to fulfill. The contract expression

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

represents the *commitment* $\text{transmit}(A_1, A_2, R, T \mid P)$ followed by the contract c. The commitment must be matched by a transmit event

$$e = \text{transmit}(v_1, v_2, r, t)$$

of resource r from agent v_1 to agent v_2 at time t where the predicate

$$P[A_1 \mapsto v_1, A_2 \mapsto v_2, R \mapsto r, T \mapsto t]$$

holds. If the event matches the commitment, the residual contract is c with A_1, A_2, R, T bound to v_1, v_2, r, t, respectively. In other words, A_1, A_2, R, T are binding variable occurrences whose scope is P and c. In this fashion the subsequent contractual obligations expressed by c may depend on the actual values in the event e.

The *contract combinators* $\cdot + \cdot, \cdot \parallel \cdot$ and $\cdot; \cdot$ are used to express choice, parallelism, and sequence, respectively. E.g., the contract

```
     transmit (vendor, customer, Y, T | T < deadline)
||   (  transmit (customer, vendor, $100, T | T < deadline)
    + (transmit (customer, vendor, $55, T | T < deadline) ;
       transmit (customer, vendor, $55, T | T < deadline + 60 days)))
```

expresses a sale of resource Y. The customer is given a choice between paying $100 before a given deadline (line 2) or just paying $55 before the deadline (line 3) and then paying $55 before 60 days after the original deadline (line 4). Both the delivery and the initial payment (whichever is chosen) must occur before the deadline, but because $\cdot \parallel \cdot$ is used, no particular order is mandated.

The language also provides facilities for defining and instantiating contract templates. The construct $f(\vec{a})$ is an instantiation of a previously defined contract template f with actual parameters $\vec{a}$. Contract templates definitions can be recursive, enabling us to express repetition using this construct. The mechanics of contract template definition and instantiation are outside the scope of this paper, but interested readers are referred to Andersen *et al.* [AEH+06] for a complete description.

Example: Sales contract A somewhat more realistic contract for simple exchanges is captured in the following contract template:

```
Sale (vendor, customer, resource, pinfo as (p, t), deadline) =
   transmit (vendor, customer, resource, T | T <= deadline) ||
   (inform (vendor, customer, (resource, pinfo), T').
     (transmit (TaxAuth, vendor, -t(resource), _ ) ||
      transmit (customer, vendor, (p + t)(resource), T''
                | T'' <= T' + 8 days)))
```

Once instantiated with a particular vendor, a customer, a resource to be delivered, the pricing of those resources, and a deadline for delivery, the contract expresses a set of legal executions: the first transmit expresses an obligation on the vendor to deliver the resource to the customer by the given deadline. The vendor must also send an invoice to the customer, which then results in an obligation by the tax autorities to collect the VAT amount for the invoiced resources and by the customer to pay the vendor the agreed-upon price, plus VAT.

Example: internal processes The term contract is suggestive of modeling certain multi-party commitments with mutual consideration, specifying who the parties are, which resources are involved, and by when they are to be transmitted.

Formally, though, our contract specifications just specify sets of event sequences: they can also be used to structure and express *internal processes* within a company and then *monitor* their execution. We can define a *universal process* as a contract that can be matched by *any* transmit, transform or inform event, as long as the agents involved are both internal agents of the subject company:

```
UniversalProcess() =
(( transmit (A, B, R, T | A <= Me, B <= Me) +
   inform (A, B, info, T | A <= Me, B <= Me) +
   transform (A, R1, R2, T | A <= Me));
 UniversalProcess()) +
 Success
```

5.3.1 Routing information

Consider a variation of the contract for the sale of Y:

```
     transmit (vendor, customer, Y, T | T < deadline)
||  (  transmit (customer, vendor, $100, T | T < deadline)
    + (transmit (customer, vendor, $55, T | T < deadline) ||
       transmit (customer, vendor, $55, T | T < deadline + 60 days)))
```

Lines 3 and 4 are now conjoined using || rather than ;. If the customer transmits $55 to the vendor before the deadline, this event can match both line 3 and line 4. Although clumsily written, the contract illustrates the need for a way to disambiguate between several possible matches. We will call such disambiguation *routing information*.

The basic idea is that all nondeterminism can be reduced to a series of routing

decisions to identify the particular commitment the event is to be matched with. We can express such a series as a sequence of routing decisions of $R = \{f, s, l, r\}$, where f (first) and s (second) indicate what choice to make when a $+$ construct is encountered, and l (left) and r (right) indicate what side of a $\|$ construct to continue on. E.g., to ensure that the early payment of \$55 is, indeed, matched to line 3, the routing information would be rsl.

5.3.2 Integration with the main architecture

With the contract language in place we can provide the remaining definitions of *ContractID*, *Contract*, and *AuxData* to integrate it with the architecture:

$$\begin{aligned} \mathit{ContractID} &= \mathit{String} \\ \mathit{Contract} &= c \\ \mathit{AuxData} &= \mathit{RoutingInformation} \\ \mathit{RoutingInformation} &= \{f, s, l, r\}^* \end{aligned}$$

Here c denotes the contract body in the syntax of the contract language.

Some contract language-specific definitions remain, namely the running contracts, C, the transition relation, $C \longrightarrow C'$, and the operators $\oplus$ and $\ominus$. These require some care to ensure that the contract language's function and variable environments are handled properly. Since these have been omitted here for the sake of simplicity, we do not delve into the details, but instead refer readers to Andersen *et al.* [AEH+06].

6 Related work

6.1 Accounting Models

Since Paccioli's Quaderno work dating back to the 15th century the dominant tradition in all financial accounting has been Double-Entry Bookkeeping (DEB) (see [WKK04] for a standard financial accounting text). However, as pointed out by McCarthy in his seminal paper introducing the Resources-Events-Agents (REA) accounting model [McC82] there are possible advantages to be reaped from other approaches. We have adopted the use of Resources, Events and Agents from REA and added our own contracts as formal, structured processes. REA is sometimes described as being in contrast with DEB. Event-based accounting does not preclude the use of the ideas presented here in conjunction with DEB, however.

In the last two decades new management accounting methods have been proposed; notably activity-based costing (ABC) [ABKY01]. A distinctive feature of our model is the separation of events and interpretation – something which is not found in DEB – and this facilitates management accounting, because one does find oneself get locked into a specific method of interpretation (say, FIFO valuation). A key aspect of our architecture is that it, accordingly, separates events representing real-world (ordinarily incontrovertible) events from interpretation (such as the various valuation method employed), which are defined as report functions. Multiple interpretations can coexist, such as tax-based depreciation and internal depreciation schemes. New interpretations can be added as report functions at any time. Conversely, a report function that is no longer of interest leaves no garbage data behind.

6.2 Contract management

Contract management is a term broadly applied to concepts, models and systems for managing contractual agreements throughout their lifecycle, from negotation through creation to execution and termination There are numerous papers that investigate organizational, system integration and, to a lesser degree, semantic aspects of contract management [Bou02,SLO06].

There are also a good number of commercial IT-applications that support contract management. [7]

Judging by their descriptions these systems are primarily aimed at supporting the reliable production of *contracts as natural-language documents* and maintaining some key information about them. They do not seem to contain an expressive, yet declarative *formal* language for user-defined contract

[7] Here is a sample of contract management software in arbitrary order, without prejudice and without any claim as to completeness or representation: TotalContracts (www.procuri.com), Livelink ECM–eDOCS for Contract Management (www.opentext.com), Meridian (www.meridiansystems.com), CompleteSource Contract Management (www.moai.com), UpsideContract (www.upsidesoft.com), StatsLog4 (www.statslog.com) for construction contracts, Contraxx (www.ecteon.com), Salesforce.com (www.salesforce.com), SAP xApp Contract Lifecycle Management (www.sap.com), 8over8.com (www.8over8.com) for oil and gas contracts, IntelliContract (www.intellicontract.com), Softrax (www.softrax.com), On Demand Contract Management (www.ketera.com), Contract Assistant (www.blueridgesoftware.bz), Autotask (www.autotask.com), Contract Web (cobblestonesystems.com), Accruent cmSuite (www.accruent.com), Memba Context (www.memba.com), Emptoris Enterprise Contract Management (www.emptoris.com), Contract Advantage (www.greatminds-software.com). Please note that trademark notices have been omitted for readability.

templates, nor a theory (or tools for) correct transformation and analysis. In particular, they are generally advertised as *integrating* with ERP systems, but not, as proposed here, as *being* at their core.

6.3 Process languages

A core component of the architecture is the explicit representation of contracts or, more generally, processes that model legal/acceptable sequences of events, which are time-stamped transfers of resources and information between companies and their actual or virtual parts.

Since the seminal publications on *business process reengineering* in the early 90s [Ham90,DS90] there has been a marked shift towards a process-oriented view of the world in management science. This has naturally induced an interest in *process-aware information systems* [DvdAtH05] and enterprise process modeling [DKKS04]. A large part of this interest was devoted to various ways of expressing business processes, or—since they are commonly seen as an instance hereof—workflows. This has lead to efforts to express workflows in Petri nets [vdAHKB03], π-calculus [Ste05a,Ste05b], and a variety of other formalisms. A significant other strand of research was that of integrating processes—however they may be represented formally—with existing information systems, most saliently ERP systems. The *ARIS* framework is an important example of this [Sch00a]. Both commercial and open source ERP system (such as, SAP and *Compiere*, respectively) have introduced process concepts. As of now, however, no other ERP system has been based on processes from *first principles* to our knowledge. In other words there has been significant research of business process reengineering, workflow systems, process-aware information systems, and how to build process *on top of* ERP systems. This has lead to a consensus that processes are a useful way to think about how business operate, and a further consensus that information systems need to closely mirror the business—this is often referred to as *business/IT-alignment*. However, no publications have attempted to revise the standard ERP system architecture to directly accommodate a process-oriented view of the world as we have done here.

As said numerous formalisms exist for specifying processes, and a number of them have been applied to modeling contracts in the business domain. Timed finite state systems such as timed automata [AD94] enhance the corresponding finite state system with deadline constraints on state transitions. Careful limitation of the expressive power of the timing constraints combined with the finite-state nature enable powerful model checking techniques.

Daskalopulu demonstrates how model checking can be applied to a sales con-

tract whose interactions are modeled as a timed Petri Net [Das00]. Molina-Jimenez *et al.* show how contracts represented as finite state machines can be monitored during execution [MJSSW03]. To ensure the finite-state property of the space of control states, their *data* components—the actual *resources* exchanged—are removed, however. Control dependency of interactions on data must be abstracted in the model when rendering it as timed finite state system. In particular, simple data-dependent protocols such as payment by installment—pay as often as necessary until the amount due is paid up as long as they all occur within a certain deadline—must be approximated in some fashion in the model. The same argument seems to apply to Event-driven Process Chains (EPC) [vdA99,Kin03] and other workflow languages with event-driven transitions on a finite set of control states. Again, at that level of modeling, they factor out the data into a separate part and treat events as atomic data with no internal structure.

An interesting recent development is the explicit declarative representation of the Deontic notions [McN06] of permissions and obligations by Pace, Priscariu, and Schneider [PS07,PPS07] for declaratively representing contractual relations. What makes this work engaging is that they demonstrate an "isomorphic" translation of contractual stipulations to the formalization in their language CL; and that CL still can be subjected to model checking.

In contrast to the above finite state systems the contract language of Andersen et al. [AEH$^+$06] employed here models not only the finite state control structure of contracts, but also *all* the relevant data: the actions in our state transitions are events that carry full data representations of resources and agents and constitute thus, by themselves, an infinite domain. The data part is not factored into unspecified off-process database updates with no data-dependent control state transitions. Unsurprisingly this makes the complexity of semantically faithful contract analysis hard: equivalence with Fail, the impossible-to-satisfy contract, is NP-complete for contracts without recursion [Nis05], and it is undecidable [Nis07] with full recursion, even when restricted to a very simple predicate language with deadlines as in timed automata. We expect the paucity of the control constructs—sequential and parallel composition; recursion— however to enable practically useful analyses that include precise analysis of the resource flows. The contract language has been developed as a match for ERP systems. An empirical evaluation in that domain is future work. We expect to find the need for variations on and extensions to it, specifically support for parallel composition of contractual commitments whose interdependencies are expressed declaratively by constraints, which is why it has deliberately been designed to be a minimal core language.

It is important to observe that the contract language specifies process *types* in the sense of protocols or behavioral types, rather than executable systems. As such it is more basic than, but analogous to, the "global" Web Services Chore-

ography Description Language (WS-CDL [8]) rather than a "local" orchestration (executable process) language such as Web Services Business Process Execution Language (WS-BPEL [9]). The global communication perspective in our contract language is motivated by and inherited from the application domain, specifically the REA accounting model (see above); it constitutes, as such, a "natural" way of formulating processes in that domain. Such a global language with a *well-defined formal semantics* enables an automatic, provably correct transformation to the (parallel) subprocesses of the individual agents (partners, roles) in a process, as has been demonstrated by Carbone, Honda and Yoshida [CHY07] for an expressive WS-CDL-like language. We believe this to be an important enabling step in generating process-specific role-based user interfaces, which are expected to be important in future ERP systems.

6.4 Event-driven architectures

An event-driven architecture is any architecture that is built on the notion of components reacting to events and generating events.

As such any *run-time monitoring/verification* system can be thought of as an event-driven (sub)system. This includes active databases [RG03, Section 5.8], security automata [Sch00b,SMH01], policy engines [GRF06], access control/resource monitors, etc., whether based on automata specifications, temporal logics such as LTL [KPA03,BLS08], or, for that matter, low-level code that implements state transitions.

What makes our architecture a *process-oriented* event-driven *architecture* is that each process (contract specification) is a denotation for a set of expected event sequences. It furthermore offers a syntax for composing and subsequently automatically manipulating/transforming such denotations: run-time events are matched against events and transformed to represent the residual process, and residual processes can be input as data to—in principle arbitrary—analysis functions.

Complex Event Processing [LF98] also relies on the event view of the world, but is primarily intended for the purpose of monitoring events from several layers in a network by installing predicates and aggregators. In contrast, in our architecture events are matched against contracts that *prescribe* the expected arrival of events and act as run-time monitors. Simultanously their syntactic representation can be used as inputs to analyses for generating information, including new events for matching.

[8] `http://www.w3.org/TR/ws-cdl-10` retrieved on June 10th 2008.
[9] `http://www.oasis-open.org/committees/wsbpel` retrieved on June 10th 2008.

7 Conclusions

We have presented an event-driven architecture consisting of an event processing engine that matches economic and information events against their process (contract) specifications, and user-definable functions providing information on the state of the system. As we have seen, these functions can be specified compactly in set notation, and the specifications map closely to the actual reports in F# code.

At any given point in time the state of the system consists of the logged events and residual contracts modeling expected future events. Reports can be defined as arbitrary functions. In this fashion derived data, expressed as report functions, are strictly separated from base data, which model real-world events. Theory and technology exist for turning naively formulated functions that read the whole state each time they are executed into efficient on-line algorithms [Bri05,NL08].

The explicit representation of contracts enables defining reporting functions, ranging from useful to-do lists to, as demonstrated by Peyton-Jones and Eber [JE03], sophisticated financial valuations. By formulating analyses for all possible (specifiable) contracts, it is possible to break the binding time dependencies that normally require that a process be coded up first before a corresponding set of specific reports can be (hand-)coded for it. Notably, we believe that role-specific user interfaces can be generated from process specifications that always reflect the present state of a process, even if changed at run-time.

Formal contract specifications thus build the core of a *process-oriented event-driven architecture*: contracts function as behavioral types for event traces. In the architecture an event is matched against a user-specified contract specification to validate the contractual validity of the event and compute the residual obligations as an explicit contract specification in its own. Contract specifications cannot only be used in this passive fashion of matching and flagging errors, but are likely to be most useful as input to functions operating on them, ranging from to-do lists via automatically generated user interfaces to sophisticated stochastic analyses for financial valuation purposes.

Acknowledgments

The above reflects ongoing work within the 3rd generation Enterprise Resource Planning Systems Project (3gERP.org), a collaboration between Copenhagen Business School, the University of Copenhagen and Microsoft Development Center Copenhagen, made possible by a grant by the Danish National Ad-

vanced Technology Foundation.

References

[ABKY01] Anthony A. Atkinson, Rajiv D. Banker, Robert S. Kaplan, and S. Mark Young. *Management Accounting.* Prentice Hall, third internation edition edition, 2001.

[AD94] Rajeev Alur and David Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, April 1994.

[AEH+06] Jesper Andersen, Ebbe Elsborg, Fritz Henglein, Jakob Grue Simonsen, and Christian Stefansen. Compositional specification of commercial contracts. *International Journal on Software Tools for Technology Transfer (STTT)*, 8(6):485–516, November 2006.

[BLS08] Andreas Bauer, Martin Leucker, and Christian Schallhart. The good, the bad, and the ugly—but how ugly is ugly? Technical Report TUM-I0803, Institut für Informatik, Technische Universität München, February 2008.

[Bou02] Abdel Boulmakoul. Integrated contract management. Technical report, Hewlett-Packard Laboratories Bristol, July 2002. http://www.hpl.hp.com/techreports/2002/HPL-2002-183.pdf.

[Bri05] Daniel Brixen. Incremental methods for REA-based reporting. M.S. thesis, Department of Computer Science, University of Copenhagen, January 2005. In Danish.

[CHY07] Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured communication-centred programming for web services. In *Proc. 16th European Symposium on Programming (ESOP), Braga, Portugal*, volume 4421 of *Lecture Notes in Computer Science (LNCS)*, pages 2–17. Springer, 2007.

[Cre75] Michael A. Crew. *Theory of the Firm.* Longman, 1975.

[Das00] Aspassia Daskalopulu. Model checking contractual protocols. In Winkels Breuker, Leenes, editor, *13th Annual Conference on Legal Knowledge and Information Systems (JURIX)*, Frontiers in Artificial Intelligence and Applications, pages 35–47. IOS Press, 2000.

[DKKS04] Nikunj P. Dalal, Manjunath Kamath, William J. Kolarik, and Eswar Sivaraman. Toward an integrated framework for modeling enterprise processes. *Commun. ACM*, 47(3):83–87, 2004.

[DS90] Th. H. Davenport and J. E. Short. The new industrial engineering: Information technology and business process reengineering. *Sloan Management Review*, 31(4), 1990.

[DvdAtH05] Marlon Dumas, Wil M. van der Aalst, and Arthur H. ter Hofstede. *Process Aware Information Systems: Bridging People and Software Through Process Technology.* Wiley-Interscience, 2005.

[GRF06] Pedro Gama, Carlos Ribeiro, and Paulo Ferreira. A scalable history-based policy engine. In *7th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY)*, pages 100–112. IEEE Computer Society, 2006.

[Ham90] M. Hammer. Reengineering work: Don't automate, obliterate. *Harvard Business Review*, 68(4):104ff, 1990.

[JE03] Simon Peyton Jones and Jean-Marc Eber. How to write a financial contract. In Gibbons and de Moor, editors, *The Fun of Programming.* Palgrave Macmillan, 2003.

[Kin03] 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.

[KPA03] Kåre J. Kristoffersen, Christian Pedersen, and Henrik R. Andersen. Runtime verification of timed ltl using disjunctive normalized equation systems. In *Proc. 3d Workshop on Runtime Verification (RV), Boulder, Colorado*, volume Electronic Notes in Theoretical Computer Science, vol. 9, no. 2, 2003.

[LF98] David C. Luckham and Brian Frasca. Complex event processing in distributed systems. Technical report, Computer Systems Lab, Stanford University, August 1998.

[LN07] Ken Friis Larsen and Michael Nissen. FunSETL—functional reporting for ERP systems. In *Proc. 19th International Symposium on Implementation and Application of Functional Languages (IFL)*, Freiburg, Germany, October 2007.

[McC82] 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.

[McN06] Paul McNamara. Deontic logic. Stanford Encyclopedia of Philosophy, 2006.

[MJSSW03] Carlos Molina-Jimenez, Santosh Shrivastava, Ellis Solaiman, and John Warne. Contract representation for run-time monitoring and enforcement. In *Proc. 2003 IEEE International Conference on E-Commerce Technology (CEC*, page 103 ff., 2003. Technical Report CS-TR-810, School of Computing Science, University of Newcastle upon Tyne.

[Nis05] Michael Nissen. B.S. thesis, January 2005. Department of Computer Science, University of Copenhagen (DIKU).

[Nis07] Michael Nissen. Contract analysis. TOPPS D-Report D-564, Department of Computer Science, University of Copenhagen (DIKU), June 2007.

[NL08] Michael Nissen and Ken Friis Larsen. FunSETL—functional reporting for ERP systems. In *Proceedings of The Ninth Symposium on Trends in Functional Programming (TFP)*, The Netherlands, May 2008.

[PPS07] Gordon Pace, Cristian Prisacariu, and Gerardo Schneider. Model checking contracts –a case study. In *Proc. 5th International Symposium on Automated Technology for Verification and Analysis (ATVA'07)*, volume 4762 of *Lecture Notes in Computer Science (LNCS)*, pages 82–97, Tokyo, Japan, October 2007. Springer-Verlag.

[PS07] Cristian Prisacariu and Gerardo Schneider. A formal language for electronic contracts. In *Proc. 9th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'07)*, volume 4468 of *Lecture Notes in Computer Science (LNCS)*, pages 174–189, Paphos, Cyprus, June 2007. Springer.

[RG03] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill, 3d edition, 2003.

[Sch00a] August-Wilhelm Scheer. *Aris-Business Process Modeling*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2000.

[Sch00b] Fred B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, 2000.

[SDDS86] J. Schwartz, R. Dewar, E. Dubinsky, and E. Schonberg. *Programming with Sets: An Introduction to SETL*. Springer-Verlag, 1986.

[SGC07] Don Syme, Adam Granicz, and Antonio Cisternino. *Expert F#*. Apress, 2007.

[SLO06] Jan W. Schemm, Christine Legner, and Hubert Österle. E-contracting: Towards electronic collaboration processes in contract management. In Franz Lehner, Holger Nösekabel, and Peter Kleinschmidt, editors, *Proc. Multikonferenz Wirtschaftsinformatik 2006*, volume Tagungsband 1, pages 255–272, 2006. http://www.gbv.de/dms/tib-ub-hannover/508826985.pdf.

[SMH01] Fred Schneider, Greg Morrisett, and Robert Harper. A language-based approach to security. In *Informatics—10 years back, 10 years ahead*, volume 2000 of *Lecture Notes in Computer Science (LNCS)*, pages 86–101. Springer, 2001.

[Ste05a] 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.

$$
\begin{aligned}
PaymentsReceived = \{&(i, USD(R)) \\
&: (transmit(A, B, R), t, i) \in Events \mid B \leq me\} \\
CashReceived = &\sum\{k : (i, k) \in PaymentsReceived\} \\
PaymentsMade = \{&(i, USD(R)) \\
&: (transmit(A, B, R), t, i) \in Events \mid A \leq me\} \\
CashPaid = &\sum\{k : (i, k) \in PaymentsMade\} \\
NetCashFlow = &\, CashReceived - CashPaid
\end{aligned}
$$

$$
\begin{aligned}
FAssetAcq = Sort\{&(R, (priceInfo, t)) \\
&: (transmit(me, me.FixedAssets, R), t, i) \in Events, \\
&(A, R', priceInfo) = InvoicesReceived(i)\} \\
Expenses = Sort\{&(R, (priceInfo, t)) \\
&: (transmit(me, me.Expenses, R), t, i) \in Events, \\
&(A, R', priceInfo) = InvoicesReceived(i)\}
\end{aligned}
$$

$$
\begin{aligned}
VATOutgoing &= \sum\{m(R) : (i, (A, R, ((p, m), t))) \in InvoicesSent\} \\
VATIncoming &= \sum\{m(R) : (i, (A, R, ((p, m), t))) \in InvoicesReceived\}
\end{aligned}
$$

Fig. A.1. Definitions of more subreports

[Ste05b] 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.

[vdA99] W.M.P. van der Aalst. Formalization and verification of event-driven process chains. *Information and Software Technology*, 41(10):639–650, July 1999.

[vdAHKB03] W. M. P. van der Aalst, A. H. M. Ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow patterns. *Distrib. Parallel Databases*, 14(1):5–51, 2003.

[WKK04] Jerry J. Weygandt, Donald E. Kieso, and Paul D. Kimmel. *Financial Accounting, with Annual Report.* Wiley, 2004.

A Subreport descriptions

Depreciation An interesting aspect is depreciation of fixed assets because it reflects certain accounting assumptions, whether mandated by law or to

reflect realistic wear-and-tear or resale considerations. For example, in the *straight line depreciation method* the depreciation per time is the same over the useful life-time of an asset; in other words, the value of a resource is a linear function of time over its depreciation period, thereafter it is 0. The *declining balance depreciation method* writes down value of priced resources by a certain percentage after each equally long period, with special write-down to 0 once the value has become sufficiently small; in other words, it is a discrete (periodicized) approximation to an exponential decay function.

We can model such depreciation as report functions or even higher-order functions on valuations. Given a priced resource $(R, ((p, m), t))$, and a depreciation period d, valuation $p_{t'}$ of R at time t' according to the straight line method such that $t \leq t' \leq t + d$ is $\frac{t'-t}{d}p$. The declining balance method can also be modeled as a time-dependent scaling of a resource's purchase valuation, with the notable exception of the final write-down to zero, since that step does not distribute over set union of priced resources. This can be handled by associating the write-down-to-0 threshold with the internal agent containing the resources, such as *FixedAssets*: The value of the priced resources at time t is computed according the depreciation formula in use (linear or exponential), and then the total value is compared to the threshold value associated with the internal agent whose resources are being value. If it is above the threshold value, its value is returned; otherwise 0 is returned.

The *NetCashFlow* **report** Cash-flow can be simply given by gathering up all money flows in the events.

Here *USD* is the base vector (as a linear function) for a currency resource. Multi-currency cash flow can be performed by taking the sum of all such base vectors; for instance $USD + Euro + DKK$. [10]

The *FAssetAcq* **and** *Expenses* **reports** Analogously to the *inventory acquisitions* we define *fixed asset acquisitions* and *operational expenses*. Note that the internal agent *Expenses* stands for resources that are consumed instantaneously.

VATOutgoing **and** *VATIncoming* The incoming and outgoing VAT amounts are computed from the invoices by employing their VAT valuation components.

[10] Recall that currencies such as USD, Euro, DKK are treated as regular resource names.