Compositional Specification of Commercial Contracts

Jesper Andersen, Ebbe Elsborg, Fritz Henglein, Jakob Grue Simonsen, and Christian Stefansen

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

Abstract. We present a declarative language for compositional specification of contracts governing the exchange of resources. It extends Eber and Peyton Jones's declarative language for specifying financial contracts to the exchange of money, goods and services amongst multiple parties and complements McCarthy's Resources/Events/Agents (REA) accounting model with a view-independent formal contract model that supports definition of user-defined contracts, automatic monitoring under execution, and user-definable analysis of their state before, during and after execution. We provide several realistic examples of commercial contracts and their analysis. A variety of (real) contracts can be expressed in such a fashion as to support their integration, management and analysis in an operational environment that registers events.

1 Introduction

When entrepreneurs enter contractual relationships with a large number of other parties, each with possible variations on standard contracts, they are confronted with the interconnected problems of *specifying* contracts, *monitoring* their execution for performance[1], *analyzing* their ramifications for planning, pricing and other purposes prior to and during execution, and *integrating* this information with accounting, workflow management, supply chain management, production planning, tax reporting, decision support *etc*.

1.1 Problems with Informal Contract Management

Typical problems that can arise in connection with informal modeling and representation of contracts and their execution include: (i) disagreement on what a contract actually requires; (ii) agreement on contract, but disagreement on what events have actually happened (event history); (iii) agreement on contract and event history, but disagreement on remaining contractual obligations; (iv) breach or malexecution of contract; (v) entering bad or undesirable contracts/missed opportunities; (vi) bad coordination of contractual obligations with production planning and supply chain management; (vii) impossibility, slowness or costliness in evaluating state of company affairs.

Anecdotal evidence suggests that costs associated with these problems can be considerable. Eber estimates that a major French investment bank has costs of about 50 mio. Euro per year attributable to (i) and (iv) above, with about half due to legal costs in connection with contract disputes and the other half due to malexecution of financial contracts [Ebe02].

In summary, capturing contractual obligations precisely and managing them conscientiously is important for a company's planning, evaluation, and reporting to management, shareholders, tax authorities, regulatory bodies, potential buyers, and others.

We argue that a declarative *domain-specific (specification) language (DSL)* for compositional specification of commercial contracts (defining contracts by combining subcontracts in various, well-defined ways) with an associated precise *operational semantics* is ideally suited to alleviating the above problems.

[1] *Performance* in contract lingo refers to *compliance* with the *promises* (contractual commitments) stipulated in a contract; nonperformance is also termed *breach of contract*.

1.2 Contributions

We (i) extend the contract language of Peyton-Jones, Eber and Seward for two-party financial contracts in a view-independent fashion to multi-party commercial contracts with iteration and first-order recursion. They involve explicit agents and transfers of arbitrary resources (money, goods and services, or even pieces of information), not only currencies. Our contract language is stratified into a pluggable base language for atomic contracts (commitments) and a combinator language for composing commitments into structured contracts. In addition, we (ii) provide a natural contract semantics based on an inductive definition for when a trace—a finite sequence of events—constitutes a successful ("performing") completion of a contract. This induces a denotational semantics, which compositionally maps contracts to trace sets as in Hoare's Communicating Sequential Processes (CSP). We (iii) systematically develop three operational semantics in a stepwise fashion, starting from the denotational semantics: A reduction semantics with deferred matching of events to specific commitments in a contract; an eager matching semantics in which events are matched nondeterministically against commitments; and finally an eager matching semantics where an event is equipped with explicit control information that *routes* it deterministically to a particular commitment. Finally, we (iv) validate applicability of our language by encoding a variety of existing contracts in it, and illustrate analyzability of contracts by providing examples of compositional analysis.

Our work builds on a previous language design by Andersen and Elsborg [AE03] and is inspired by Peyton Jones and Eber's compositional specification of financial contracts, the REA accounting model and CSP-like process algebras. See Section 7 for a comparison with that work.

2 Modeling Commercial Contracts

A *contract* is an agreement between two or more parties which creates obligations to do or not do the specific things that are the subject of that agreement. A *commercial contract* is a contract whose subject is the exchange of scarce *resources* (money, goods, and services). Examples of commercial contracts are sales orders, service agreements, and rental agreements. Adopting terminology from the REA accounting model [McC82] we shall also call obligations *commitments* and parties *agents*.

2.1 Contract Patterns

In its simplest form a contract commits two contract parties to an exchange of resources such as goods for money or services for money; that is to a pair of *transfers* of resources from one party to the other, where one transfer is in *consideration* of the other.

The sales order *template* in Figure 1 commits the two parties (`seller`, `buyer`) to a pair of transfers, of `goods` from `seller` to `buyer` and of `money` from `buyer` to `seller`. Many commercial contracts are of this simple *quid-pro-quo* kind, but far from all. Consider the legal services agreement template in Figure 2. Here commitments for rendering of a monthly legal service are *repeated*, and each monthly service consists of a standard service part and an *optional* service part. More generally, a contract may allow for *alternative* executions, any one of which satisfies the given contract.

We can discern the following basic *contract patterns* for composing commercial contracts from subcontracts (a subcontract is a contract used as part of another contract):

- a *commitment* stipulates the transfer of a resource or set of resources between two parties; it constitues an *atomic contract*;
- a contract may require *sequential* execution of subcontracts;
- a contract may require *concurrent* execution of subcontracts, that is execution of all subcontracts, where individual commitments may be interleaved in arbitrary order;

– a contract may require execution of one of a number of *alternative* subcontracts;
– a contract may require *repeated* execution of a subcontract.

In the remainder of this paper we shall explore a declarative contract specification language based on these contract patterns.

Fig. 1 Agreement to Sell Goods

Section 1. (Sale of goods) Seller shall sell and deliver to buyer (description of goods) no later than (date).
Section 2. (Consideration) In consideration hereof, buyer shall pay (amount in dollars) in cash on delivery at the place where the goods are received by buyer.
Section 3. (Right of inspection) Buyer shall have the right to inspect the goods on arrival and, within (days) business days after delivery, buyer must give notice (detailed-claim) to seller of any claim for damages on goods.

Fig. 2 Agreement to Provide Legal Services

Section 1. The attorney shall provide, on a non-exclusive basis, legal services up to (n) hours per month, and furthermore provide services in excess of (n) hours upon agreement.
Section 2. In consideration hereof, the company shall pay a monthly fee of (amount in dollars) before the 8th day of the following month and (rate) per hour for any services in excess of (n) hours 40 days after the receival of an invoice.
Section 3. This contract is valid 1/1-12/31, 2004.

3 Compositional Contract Language

In this section we present a core contract specification language that reflects the contract composition patterns of Section 2.1. This is a cursory presentation, with no proofs given. See the technical report [AEH+04] for a full presentation.

3.1 Syntax

Our contract language $\mathcal{C}^{\mathcal{P}}$ is defined inductively by the inference system for deriving judgements of the forms $\Gamma; \Delta \vdash c : \mathrm{Contract}$ and $\Delta \vdash D : \Gamma$. Here Γ and Δ range over maps from identifiers to *contract template types* and to *base types*, respectively. The $\oplus$-operator on maps is defined as follows:

$$(m \oplus m')(x) = \begin{cases} m'(x) \text{ if } x \in \mathrm{domain}(m') \\ m(x) \text{ otherwise} \end{cases}$$

The language is built on top of a typed *base language* P defined by $\Delta \vdash a : \tau$ that defines expressions denoting *agents*, *resources*, *time*, other basic types and predicates (Boolean expressions) over those. P provides the possibility of referring to *observables* [JES00,JE03]. The language is parametric in P, and we shall introduce suitable base language expressions on an *ad hoc* basis in our examples for illustrative purposes.

The language $\mathcal{C}^{\mathcal{P}}$ is defined by the inference system in Figure 3. If judgement $\Gamma; \Delta \vdash c : \mathrm{Contract}$ is derivable, we say that c is a well-defined contract given type assumptions Γ and Δ. $\mathrm{Success}$ denotes the *trivial* or *(successfully) completed* contract: it carries no obligations on

Fig. 3 Syntax for contract specifications

$$\Gamma; \Delta \vdash \text{Success} : \text{Contract} \qquad \Gamma; \Delta \vdash \text{Failure} : \text{Contract}$$

$$\frac{\Gamma(f) = \boldsymbol{\tau} \to \text{Contract} \quad \Delta \vdash \boldsymbol{a} : \boldsymbol{\tau}}{\Gamma; \Delta \vdash f(\boldsymbol{a}) : \text{Contract}} \qquad \frac{\begin{array}{c}\Delta' = \Delta \oplus \{A_1 : \text{Agent}, A_2 : \text{Agent}, R : \text{Resource}, T : \text{Time}\} \\ \Gamma; \Delta' \vdash c : \text{Contract} \\ \Delta' \vdash P : \text{Boolean}\end{array}}{\Gamma; \Delta \vdash \text{transmit}(A_1, A_2, R, T \mid P).\, c : \text{Contract}}$$

$$\frac{\Gamma; \Delta \vdash c_1 : \text{Contract} \quad \Gamma; \Delta \vdash c_2 : \text{Contract}}{\Gamma; \Delta \vdash c_1 + c_2 : \text{Contract}} \qquad \frac{\Gamma; \Delta \vdash c_1 : \text{Contract} \quad \Gamma; \Delta \vdash c_2 : \text{Contract}}{\Gamma; \Delta \vdash c_1 \parallel c_2 : \text{Contract}}$$

$$\frac{\Gamma; \Delta \vdash c_1 : \text{Contract} \quad \Gamma; \Delta \vdash c_2 : \text{Contract}}{\Gamma; \Delta \vdash c_1; c_2 : \text{Contract}} \qquad \frac{\begin{array}{c}\Gamma = \{f_i \mapsto \tau_{i1} \times \ldots \times \tau_{in_i} \to \text{Contract}\}_{i=1}^{m} \\ \Gamma; \Delta \oplus \{X_{i1} : \tau_{i1}, \ldots, X_{in_i} : \tau_{in_i}\} \vdash c_i : \text{Contract}\end{array}}{\Delta \vdash \{f_i[\boldsymbol{X_i}] = c_i\}_{i=1}^{m} : \Gamma}$$

$$\frac{\Delta \vdash \{f_i[\boldsymbol{X_i}] = c_i\}_{i=1}^{m} : \Gamma \quad \Gamma; \Delta \vdash c : \text{Contract}}{\Delta \vdash \text{letrec}\, \{f_i[\boldsymbol{X_i}] = c_i\}_{i=1}^{m} \text{ in } c : \text{Contract}}$$

anybody. Failure denotes the *inconsistent* or *failed* contract; it signifies breach of contract or a contract that is impossible to fulfill. The environment $\mathrm{D} = \{f_i[\boldsymbol{X_i}] = c_i\}_{i=1}^{m}$ contains named *contract templates*. A contract template needs to be instantiated with actual arguments from the base language. The contract expression $\text{transmit}(A_1, A_2, R, T \mid P).\, c$ represents a contract where the *commitment* $\text{transmit}(A_1, A_2, R, T \mid P)$ must be satisfied first. Note that A_1, A_2, R, T are binding variable occurrences whose scope is P and c. The commitment must be *matched* by a *(transfer) event* $e = \text{transmit}(a_1, a_2, r, t)$ of resource r from agent a_1 to agent a_2 at time t where $P(a_1, a_2, r, t)$ holds. After matching, the residual contract is c in which A_1, A_2, R, T are bound to a_1, a_2, r, t, respectively. In this fashion, the subsequent contractual obligations expressed by c may depend on the actual values in event e. The *contract combinators* $\cdot + \cdot, \cdot \parallel \cdot$ and $\cdot; \cdot$ compose subcontracts according to the contract patterns we have discerned: by alternation, concurrently, and sequentially, respectively. A contract consists of a finite set of named contract templates and a contract body. Note that contract templates may be (mutually) recursive, which, in particular, lets us capture repetition of subcontracts. In the following we shall adopt the convention that A_1, A_2, R, T must not be bound in environment Δ. If a variable from Δ or any expression a only involving variables bound in Δ occurs as an argument of a transmit, we interpret this as an abbreviation; e.g., $\text{transmit}((a, A_2, R, T \mid P)).\, c$ abbreviates $\text{transmit}((A_1, A_2, R, T \mid P \wedge A_1 = a)).\, c$ where A_1 is a new (agent-typed) variable not bound in Δ and different from A_2, R and T. We abbreviate $\text{transmit}(A_1, A_2, R, T \mid P).\, \text{Success}$ to $\text{transmit}(A_1, A_2, R, T \mid P)$. Examples encoding the contracts from Figures 1 and 2 are presented in Section 4.

3.2 Event Traces and Contract Satisfaction

A contract specifies a set of alternative performing event sequences (contract executions), each of which satisfies the obligations expressed in the contract and concludes it. In this section we make these notions precise for our language.

A *base structure* is a tuple $(\mathcal{R}, \mathcal{T}, \mathcal{A})$ of sets of resources $\mathcal{R}$, agents $\mathcal{A}$ and a totally ordered set $(\mathcal{T}, \leq_{\mathcal{T}})$ of *dates* (or *time points*), plus other sets for other types, as needed. A *(transfer) event* e is a term $\text{transmit}(a_1, a_2, r, t)$, where $a_1, a_2 \in \mathcal{A}, r \in \mathcal{R}$ and $t \in \mathcal{T}$. An *(event) trace* s is a finite sequence of events that is chronologically ordered; that is, for $s = e_1 \ldots e_n$ the time points in $e_1 \ldots e_n$ occur in ascending order. We adopt the following notation: $\langle\rangle$ denotes the empty sequence; a trace consisting of a single event e is denoted by e itself; concatenation of traces

s_1 and s_2 is denoted by juxtaposition: $s_1 s_2$; we write $(s_1, s_2) \rightsquigarrow s$ if s is an interleaving of the events in traces s_1 and s_2; we write $\boldsymbol{X}$ for the vector $X_1, \ldots, X_k$ with $k \geq 0$ and where k can be deduced from the context; we write $P[a_1/A_1, a_2/A_2, r/R, t/T]$ and $c[a_1/A_1, a_2/A_2, r/R, t/T]$ for *substitution* of expressions a_1, a_2, r, t for free variables A_1, A_2, R, T in Boolean expression P and contract expression c, respectively.[2] We are now ready to specify when a trace *satisfies* a contract, i.e. gives rise to a performing execution of the contract. This is done inductively by the inference system for judgements $s \vdash^{\delta}_{\mathrm{D}} c$ in Figure 4, where $\mathrm{D} = \{f_i[\boldsymbol{X_i}] = c_i\}_{i=1}^{m}$ is a finite set of named *contract templates* and δ is a finite set of bindings of variables to elements of the given base structure. A derivable judgement $s \vdash^{\delta}_{\mathrm{D}} c$ expresses that event sequence s satisfies—successfully executes and concludes—contract c in an environment where contract templates are defined as in D and δ specifies to which values the base variables in c and D are bound. Conversely, if $s \vdash^{\delta}_{\mathrm{D}} c$ is not derivable then s does not satisfy c. The premise $\delta \models P[a_1/A_1, a_2/A_2, r/R, t/T]$ in the 3d rule stipulates that $P[a_1/A_1, a_2/A_2, r/R, t/T]$, with free variables bound as in δ, must be true for an event to match the corresponding commitment.

Fig. 4 Contract satisfaction

$$\langle\rangle \vdash^{\delta}_{\mathrm{D}} \text{Success} \qquad \frac{s \vdash^{\delta}_{\mathrm{D}} c[\boldsymbol{a}/\boldsymbol{X}] \quad (f[\boldsymbol{X}] = c) \in \mathrm{D}}{s \vdash^{\delta}_{\mathrm{D}} f(\boldsymbol{a})}$$

$$\frac{\delta \models P[a_1/A_1, a_2/A_2, r/R, t/T] \quad s \vdash^{\delta}_{\mathrm{D}} c[a_1/A_1, a_2/A_2, r/R, t/T]}{\mathrm{transmit}(a_1, a_2, r, t)\, s \vdash^{\delta}_{\mathrm{D}} \mathrm{transmit}((A_1, A_2, R, T|P)).\, c}$$

$$\frac{s_1 \vdash^{\delta}_{\mathrm{D}} c_1 \quad s_2 \vdash^{\delta}_{\mathrm{D}} c_2 \quad (s_1, s_2) \rightsquigarrow s}{s \vdash^{\delta}_{\mathrm{D}} c_1 \parallel c_2} \qquad \frac{s_1 \vdash^{\delta}_{\mathrm{D}} c_1 \quad s_2 \vdash^{\delta}_{\mathrm{D}} c_2}{s_1 s_2 \vdash^{\delta}_{\mathrm{D}} c_1 ; c_2}$$

$$\frac{s \vdash^{\delta}_{\mathrm{D}} c}{s \vdash^{\delta} \mathrm{letrec\ D\ in}\ c} \qquad \frac{s \vdash^{\delta}_{\mathrm{D}} c_1}{s \vdash^{\delta}_{\mathrm{D}} c_1 + c_2} \qquad \frac{s \vdash^{\delta}_{\mathrm{D}} c_2}{s \vdash^{\delta}_{\mathrm{D}} c_1 + c_2}$$

3.3 Contract Monitoring by Residuation

Extensionally, contracts classify traces (event sequences) into performing and nonperforming ones. We define the *extension* of a contract c to be the set of its performing executions: $\mathcal{C}[\![c]\!]^{\mathrm{D};\delta} = \{s : s \vdash^{\delta}_{\mathrm{D}} c\}$. We say c *denotes* a trace set S in context D, δ, if $\mathcal{C}[\![c]\!]^{\mathrm{D};\delta} = S$.[3]

We are not only interested in classifying complete event sequences once they have happened, though, but in *monitoring* contract execution as it unfolds in time under the arrival of events.

Given a trace set S denoted by a contract c and an event e, the *residuation function* $\cdot/\cdot$ captures how c can be satisfied if the first event is e. It is defined as follows:

$$S/e = \{s' \mid \exists s \in S : es' = s\}$$

Conceptually, we can map contracts to trace sets and use the residuation function to monitor contract execution as follows:

[2] We have not specified a particular language of Boolean expressions; we only require that it has a well-defined notion of substitution.

[3] A variant of $\mathcal{C}[\![c]\!]^{\mathrm{D};\delta}$ can be characterized compositionally, yielding a *denotational* semantics; see [AEH+04].

1. Map a given contract c_0 to the trace set S_0 that it denotes. If $S_0 = \emptyset$, stop and output "inconsistent".
2. For $i = 0, 1, \ldots$ do:
   Receive message e_i.
   (a) If e_i is a transfer event, compute $S_{i+1} = S_i/e_i$. If $S_{i+1} = \emptyset$, stop and output "breach of contract"; otherwise continue.
   (b) If e_i is a "terminate contract" message, check whether $\langle\rangle \in S_i$. If so, all obligations have been fulfilled and the contract can be terminated. Stop and output "successfully completed". If $\langle\rangle \notin S_i$, output "cannot be terminated now", let $S_{i+1} = S_i$ and continue to receive messages.

To make the conceptual algorithm for contract life cycle monitoring from Section 3.3 *operational*, we need to represent the residual trace sets and provide methods for deciding tests for emptiness and failure. In particular, we would like to use contracts as representations for trace sets. Not all trace sets are denotable by contracts, however. In particular, given a contract c that denotes a trace set S_c it is not *a priori* clear whether S_c/e is denotable by a contract c'. If it is, we call c' the *residual contract of c after e*.

3.4 Nullable and Guarded Contracts

In this section we characterize *nullability* of a contract and introduce *guarding*, which is a sufficient condition on contracts for ensuring that residuation can be performed by reduction on contracts.

Fig. 5 Nullable contracts

$$\frac{\mathrm{D} \vdash c \text{ nullable} \quad (f[\boldsymbol{X}] = c) \in \mathrm{D}}{\mathrm{D} \vdash f(\boldsymbol{a}) \text{ nullable}} \qquad \frac{\mathrm{D} \vdash c \text{ nullable}}{\mathrm{D} \vdash c + c' \text{ nullable}} \qquad \frac{\mathrm{D} \vdash c' \text{ nullable}}{\mathrm{D} \vdash c + c' \text{ nullable}}$$

$$\mathrm{D} \vdash \mathrm{Success} \text{ nullable} \qquad \frac{\mathrm{D} \vdash c \text{ nullable} \quad \mathrm{D} \vdash c' \text{ nullable}}{\mathrm{D} \vdash c \parallel c' \text{ nullable}} \qquad \frac{\mathrm{D} \vdash c \text{ nullable} \quad \mathrm{D} \vdash c' \text{ nullable}}{\mathrm{D} \vdash c; c' \text{ nullable}}$$

Let us write $\mathrm{D} \models c$ nullable if $\langle\rangle \in \mathcal{C}[\![c]\!]^{\mathrm{D};\delta}$ for all δ. We call such a contract *nullable* (or *terminable*): it can be concluded successfully, but may possibly also be continued. E.g., the contract $\mathrm{Success} + \mathrm{transmit}(a_1, a_2, r, t|P)$ is nullable, as it may be concluded successfully (left choice). Note however, that it may also be continued (right choice). It is easy to see that nullability is independent of δ: $\langle\rangle \in \mathcal{C}[\![c]\!]^{\mathrm{D};\delta}$ for some δ if and only if $\langle\rangle \in \mathcal{C}[\![c]\!]^{\mathrm{D};\delta'}$ for any other δ'. Deciding nullability is required to implement Step 2b in contract monitoring. The following proposition expresses that nullability is characterized by the inference system in Figure 5.

Proposition 1. $\mathrm{D} \models c$ nullable $\iff \mathrm{D} \vdash c$ nullable

A contract c is *(hereditarily) guarded* in context D if $\mathrm{D} \vdash c$ guarded is derivable from Figure 6; intuitively, guardedness ensures that in a contract with mutual recursion, we do not have (mutual) recursions such as $\{f[\boldsymbol{X}] = g[\boldsymbol{X}], g[\boldsymbol{X}] = f[\boldsymbol{X}]\}$ that cause the residuation algorithm to loop infinitely.

3.5 Operational Semantics I: Deferred Matching

Residuation on trace sets tells us how to maintain the trace set under arrival of events. In this section we present a *reduction semantics* for contracts, which lifts residuation on trace sets to contracts and thus provides a *monitoring semantics* for contract execution.

Fig. 6 Guarded contracts

$$\mathrm{D} \vdash \text{Success guarded} \qquad \mathrm{D} \vdash \text{Failure guarded}$$

$$\mathrm{D} \vdash \text{transmit}(\boldsymbol{X} \mid P).\, c \text{ guarded} \qquad \frac{\mathrm{D} \vdash c \text{ guarded} \quad (f[\boldsymbol{X}] = c) \in \mathrm{D}}{\mathrm{D} \vdash f(\boldsymbol{a}) \text{ guarded}}$$

$$\frac{\mathrm{D} \vdash c \text{ guarded} \quad \mathrm{D} \vdash c' \text{ guarded}}{\mathrm{D} \vdash c + c' \text{ guarded}} \qquad \frac{\mathrm{D} \vdash c \text{ guarded} \quad \mathrm{D} \vdash c' \text{ guarded}}{\mathrm{D} \vdash c \parallel c' \text{ guarded}}$$

$$\frac{\mathrm{D} \vdash c \text{ guarded} \quad \mathrm{D} \vdash c' \text{ guarded}}{\mathrm{D} \vdash c; c' \text{ guarded}}$$

Fig. 7 Deterministic reduction (delayed matching)

$$\mathrm{D}, \delta \vdash_D \text{Success} \xrightarrow{e} \text{Failure} \qquad \mathrm{D}, \delta \vdash_D \text{Failure} \xrightarrow{e} \text{Failure}$$

$$\frac{\delta \models P[\boldsymbol{a}/\boldsymbol{X}]}{\mathrm{D}, \delta \vdash_D \text{transmit}(\boldsymbol{X}|P).\, c \xrightarrow{\text{transmit}(\boldsymbol{a})} c[\boldsymbol{a}/\boldsymbol{X}]} \qquad \frac{\delta \not\models P[\boldsymbol{a}/\boldsymbol{X}]}{\mathrm{D}, \delta \vdash_D \text{transmit}(\boldsymbol{X}|P).\, c \xrightarrow{\text{transmit}(\boldsymbol{a})} \text{Failure}}$$

$$\frac{\mathrm{D}, \delta \vdash_D c[\boldsymbol{a}/\boldsymbol{X}] \xrightarrow{e} c' \quad (f[\boldsymbol{X}] = c) \in \mathrm{D}}{\mathrm{D}, \delta \vdash_D f(\boldsymbol{a}) \xrightarrow{e} c'} \qquad \frac{\mathrm{D}, \delta \vdash_D c \xrightarrow{e} d \quad \mathrm{D}, \delta \vdash_D c' \xrightarrow{e} d'}{\mathrm{D}, \delta \vdash_D c + c' \xrightarrow{e} d + d'}$$

$$\frac{\mathrm{D}, \delta \vdash_D c \xrightarrow{e} d \quad \mathrm{D}, \delta \vdash_D c' \xrightarrow{e} d'}{\mathrm{D}, \delta \vdash_D c \parallel c' \xrightarrow{e} c \parallel d' + d \parallel c'} \qquad \frac{\mathrm{D} \vdash c \text{ nullable} \quad \mathrm{D}, \delta \vdash_D c \xrightarrow{e} d \quad \mathrm{D}, \delta \vdash_D c' \xrightarrow{e} d'}{\mathrm{D}, \delta \vdash_D c; c' \xrightarrow{e} d; c' + d'}$$

$$\frac{\mathrm{D} \nvdash c \text{ nullable} \quad \mathrm{D}, \delta \vdash_D c \xrightarrow{e} d}{\mathrm{D}, \delta \vdash_D c; c' \xrightarrow{e} d; c'} \qquad \frac{\mathrm{D}, \delta \vdash_D c \xrightarrow{e} c'}{\delta \vdash_D \text{letrec D in } c \xrightarrow{e} \text{letrec D in } c'}$$

The ability of representing residual contract obligations of a partially executed contract and thus any state of a contract as a *bona fide* contract carries the advantage that any analysis that is performed on "original" contracts automatically extends to partially executed contracts as well. E.g., an investment bank that applies valuations to financial contracts before offering them to customers can apply their valuations to their portfolio of contracts under execution; e.g., to analyze its risk exposure under current market conditions.

The reduction semantics is presented in Figure 7. The basic *matching rule* is

$$\frac{\delta \models P[\boldsymbol{a}/\boldsymbol{X}]}{\mathrm{D}, \delta \vdash_D \text{transmit}(\boldsymbol{X}|P).\, c \xrightarrow{\text{transmit}(\boldsymbol{a})} c[\boldsymbol{a}/\boldsymbol{X}]}.$$

It *matches* an event with a specific commitment in a contract. There may be multiple commitments in a contract that match the same event. The semantics captures the possibilities of matching an event against multiple commitments by applying all possible reductions in alternatives and concurrent contract forms and forming the sum of their possible outcomes (some of which may actually be Failure).

The rule

$$\frac{\mathrm{D}, \delta \vdash_D c \xrightarrow{e} d \quad \mathrm{D}, \delta \vdash_D c' \xrightarrow{e} d'}{\mathrm{D}, \delta \vdash_D c + c' \xrightarrow{e} d + d'}$$

thus reduces both alternatives c and c' and then forms the sum of their respective results d, d'.

Finally, the rule

$$\frac{\mathrm{D} \vdash c \text{ nullable} \quad \mathrm{D}, \delta \vdash_D c \xrightarrow{e} d \quad \mathrm{D}, \delta \vdash_D c' \xrightarrow{e} d'}{\mathrm{D}, \delta \vdash_D c; c' \xrightarrow{e} d; c' + d'}$$

captures that e can be matched in c or, if c is nullable, in c'. Note that, if c is not nullable, e can only be matched in c, not c', as expressed by the rule

$$\frac{\mathrm{D} \nvdash c \text{ nullable} \quad \mathrm{D}, \delta \vdash_D c \xrightarrow{e} d}{\mathrm{D}, \delta \vdash_D c; c' \xrightarrow{e} d; c'}.$$

In this fashion the semantics keeps track of the results of all possible matches in a reduction sequence as explicit *alternatives* (summands) and *defers* the decision as to *which specific* commitment is matched by a particular event during contract exectution until the very end: By selecting a particular summand in a residual contract after a number of reduction steps that represents $\mathrm{Success}$ (and the contract is thus terminable) a particular set of matching decisions is chosen *ex post*. As presented, the reduction semantics gives rise to an implementation in which the multiple reducts of previous reduction steps are reduced in parallel, since they are represented as summands in a single contract, and the rule for reduction of sums reduces both summands. It is relatively straightforward to turn this into a backtracking semantics by an asymmetric reduction rule for sums, which delays reduction of the right summand.

Guardedness is key to ensuring termination of contract residuation and thus that every (guarded) contract has a residual contract under any event in the reduction semantics of Figure 7.

Theorem 1. *If $c \in \mathcal{C}^{\mathcal{P}}$ is guarded then for each event e there exists a unique $c' \in \mathcal{C}^{\mathcal{P}}$ such that $\mathrm{D}, \delta \vdash_D c \xrightarrow{e} c'$. Furthermore, we have that c' is guarded and $\mathrm{D}, \delta \models c/e = c'$, which means $\mathcal{C}[\![c]\!]^{\mathrm{D};\delta}/e = \mathcal{C}[\![c']\!]^{\mathrm{D};\delta}$.*

Using this reduction semantics we can turn our conceptual contract monitoring algorithm into a real algorithm.

Proposition 1 provides a syntactic characterization of nullability, which can easily (not trivially) be turned into an algorithm. Inconsistency—whether a contract denotes the empty trace set or not—is not treated here; see the full report [AEH+04].

3.6 Operational Semantics II: Eager Matching

The deferred matching semantics of Figure 7 is flexible and faithful to the natural notion of contract satisfaction as defined in Figure 4. But from an accounting practice point of view it is weird because matching decisions are deferred. In bookkeeping standard *modus operandi* is that events are matched against specific commitments *eagerly*; that is online, as events arrive.[4]

We shall turn the deferred matching semantics of Figure 7 into an eager matching semantics (Figure 8). The idea is simple: Represent here-and-now choices as alternative *rules* (meta-level) as opposed to alternative contracts (object level). Specifically, we split the rules for reducing alternatives and concurrent subcontracts into multiple rules, and we capture the possibility of reducing in the second component of a sequential contract by adding τ-transitions, which "spontaneously" (without a driving external event) reduce a contract of the form $\mathrm{Success}; c$ to c. For this to be sufficient we have to make sure that a nullable contract indeed can be reduced to $\mathrm{Success}$, not just a contract that is *equivalent* with $\mathrm{Success}$, such as $\mathrm{Success} \parallel \mathrm{Success}$. This is done by ensuring that τ-transitions are strong enough to guarantee reduction to $\mathrm{Success}$ as required.

[4] There are standard accounting practices for changing such decisions, but both default and standard conceptual model are that matching decisions are made as early as possible. In general, it seems representing and deferring choices and applying *hypothetical* reasoning to them appears to be a rather unusual phenomenon in accounting.

Fig. 8 Nondeterministic reduction (eager matching)

$$\mathrm{D}, \delta \vdash_N \mathrm{Success} \xrightarrow{e} \mathrm{Failure} \qquad \mathrm{D}, \delta \vdash_N \mathrm{Failure} \xrightarrow{e} \mathrm{Failure}$$

$$\frac{\delta \models P[\boldsymbol{a}/\boldsymbol{X}]}{\mathrm{D}, \delta \vdash_N \mathrm{transmit}(\boldsymbol{X} \mid P).\, c \xrightarrow{\mathrm{transmit}(\boldsymbol{a})} c[\boldsymbol{a}/\boldsymbol{X}]} \qquad \frac{\delta \not\models P[\boldsymbol{a}/\boldsymbol{X}]}{\mathrm{D}, \delta \vdash_N \mathrm{transmit}(\boldsymbol{X} \mid P).\, c \xrightarrow{\mathrm{transmit}(\boldsymbol{a})} \mathrm{Failure}}$$

$$\frac{(f[\boldsymbol{X}] = c) \in \mathrm{D}}{\mathrm{D}, \delta \vdash_N f(\boldsymbol{a}) \xrightarrow{\tau} c[\boldsymbol{a}/\boldsymbol{X}]} \qquad \mathrm{D}, \delta \vdash_N c + c' \xrightarrow{\tau} c \qquad \mathrm{D}, \delta \vdash_N c + c' \xrightarrow{\tau} c'$$

$$\frac{\mathrm{D}, \delta \vdash_N c \xrightarrow{\lambda} d}{\mathrm{D}, \delta \vdash_N c \parallel c' \xrightarrow{\lambda} d \parallel c'} \qquad \frac{\mathrm{D}, \delta \vdash_N c' \xrightarrow{\lambda} d'}{\mathrm{D}, \delta \vdash_N c \parallel c' \xrightarrow{\lambda} c \parallel d'}$$

$$\mathrm{D}, \delta \vdash_N \mathrm{Success} \parallel c \xrightarrow{\tau} c \qquad \mathrm{D}, \delta \vdash_N c \parallel \mathrm{Success} \xrightarrow{\tau} c \qquad \mathrm{D}, \delta \vdash_N \mathrm{Success}; c' \xrightarrow{\tau} c'$$

$$\frac{\mathrm{D}, \delta \vdash_N c \xrightarrow{\lambda} d}{\mathrm{D}, \delta \vdash_N c; c' \xrightarrow{\lambda} d; c'} \qquad \frac{\mathrm{D}, \delta \vdash_N c \xrightarrow{e} c'}{\delta \vdash_N \mathrm{letrec}\, \mathrm{D}\, \mathrm{in}\, c \xrightarrow{e} \mathrm{letrec}\, \mathrm{D}\, \mathrm{in}\, c'}$$

Based on these considerations we arrive at the reduction semantics in Figure 8, where meta-variable λ ranges over events e and the internal event τ. Note that it is nondeterministic and not even confluent: A contract c can be reduced to two different contracts by the same event. Consider e.g., $c = a; b + a; b'$ where a, b, b' are commitments with suitable D, δ, no two of which match the same event. For event e matching a we have $\mathrm{D}, \delta \vdash_N c \xrightarrow{e} b$ and $\mathrm{D}, \delta \vdash_N c \xrightarrow{e} b'$, but neither b nor b' can be reduced to $\mathrm{Success}$ or any other contract by the same event sequence. In reducing c we have not only resolved it against e, but also made a *decision*: whether to apply it to the first alternative of c or to the second. Technically, the reduction semantics is not closed under residuation: Given c and e it is not always possible to find c' such that $\mathrm{D}, \delta \vdash_N c \xrightarrow{e} c'$ and $\mathrm{D}; \delta \models c/e = c'$. It is sound, however, in the sense that the reduct always denotes a subset of the residual trace set:

Proposition 2. *1. If* $\mathrm{D}, \delta \vdash_N c \xrightarrow{e} c'$ *then* $\mathrm{D}, \delta \models c' \subseteq c/e$.
2. If $\mathrm{D}, \delta \vdash_N c \xrightarrow{\tau} c'$ *then* $\mathrm{D}, \delta \models c' \subseteq c$.

Even though individual eager reductions do not preserve residuation, the set of all reductions does so:

Proposition 3. *If* $\mathrm{D}, \delta \vdash_D c \xrightarrow{e} c'$ *then there exist contracts* $c_1, \ldots, c_n$ *for some* $n \geq 1$ *such that* $\mathrm{D}, \delta \vdash_N c \xrightarrow{\tau*} c_i'' \xrightarrow{e} c_i$ *for all* $i = 1 \ldots n$ *and* $\mathrm{D}, \delta \models c' \subseteq \sum_{i=1}^{n} c_i$. *The notation* $\cdot \xrightarrow{\tau*} \cdot$ *indicates any number* ≥ 0 *of* τ*-transitions.*

As a corollary, Propositions 2 and 3 combined yield that the object-level nondeterminism (expressed as contract alternatives) in the deferred matching semantics is faithfully reflected in the meta-level nondeterminism (expressed as multiple applicable rules) of the eager matching semantics.

3.7 Operational Semantics III: Eager Matching with Explicit Routing

Consider the following execution model for contracts: Two or more parties each have a copy of the contract they have previously agreed upon and monitor its execution under the arrival of events. Even though they agree on prior contract state and the next event, the parties may

arrive at different residual contracts and thus different expectations as to the future events allowed under the contract. This is because of nondeterminacy in contract execution with eager matching; e.g., a payment of $50 may match multiple payment commitments, and the parties may make different matches. We can remedy this by making *control* of contract reduction with eager matching explicit in order to make reduction deterministic: events are accompanied by control information that unambiguously prescribes how a contract is to be reduced. In this fashion parties that agree on what events have happened and on their associated control information, will reduce their contract identically. See the full technical report for details [AEH+04].

4 Example Contracts

For the purpose of demonstration we will afford ourselves a fairly advanced predicate language with basic arithmetic, logical connectives, lists and basic functions. The syntax is standard and straightforward, and the details will be obvious from the examples.

Consider the validity period specified in Section 3 of the Agreement to Provide Legal Services (Figure 2). Taken literally, it would imply, that the attorney shall render services in the month of December, but receive no fee in consideration since January 2005 is outside the validity period. Surely, this is not the intention; in fact, consideration will defeat most deadlines as is clearly the intent here. In the coding of the Agreement to Provide Legal Services the expiration date `end` has to be pushed down on all transmits despite its global nature to make sure that consideration would not be cut off.

The Agreement to Provide Legal Services fails to specify who decides if legal services should be rendered. In the coding it is simply assumed that the attorney is the initiator and that all services rendered over a month can be modelled as one event. Furthermore, the attorney is assumed to give the notice `nowork` if no work was done for the past month. This is an artifact introduced to guard the recursive call to `legal`.

Fig. 9 Software Development Agreement

Section 1. The Developer shall develop software as described in Exhibit A (Requirements Specification) according the schedule set forth in Exhibit B (Project Schedule and Deliverables). Specifically, the Developer shall be responsible for the timely completion of the deliverables identified in Exhibit B.

Section 2. The Client shall provide written approval upon the completion of each deliverable identified in Exhibit B.

Section 3. In the event of any delay by the Client, all the Developer's remaining deadlines shall be extended by the greater of the two following: (i) five working days, (ii) two times the delay induced by the Client. The Client's deadlines shall be unchanged.

Section 4. In consideration of services rendered the Client shall pay USD $100.000 due on 7/1.

Section 5. If the Client wishes to add to the order, or if upon written approval of a deliverable, the Client wishes to make modifications to the deliverable, the Client and the Developer shall enter into a Change Order. Upon mutual agreement the Change Order shall be attached to this contract.

Section 6. The Developer shall retain all intellectual rights associated with the software developed. The Client may not copy or transfer the software to any third party without the explicit, written consent of the Developer.

Exhibit A. (omitted)

Exhibit B. Deadlines for deliverables and approval: (i) 1/1, 1/15; (ii) 3/1, 3/15, (final deadline) 7/1, 7/15.

Now consider the more elaborate Software Development Agreement in Figure 9. When coding the contract, one notices that the contract fails to specify the ramifications of the client's

Fig. 10 Specification of Software Development Agreement – note that we assume (easily defined) abbreviations for `max(x,y)` and allow subtraction on the domain Time.

```
letrec
    deliverables (dev, client, payment, deliv1, deadline1, approv1,
                                        deliv2, deadline2, approv2,
                                        delivf, deadlinef, approvf) =
      transmit(dev, client, deliv1, T1 | T1 <= deadline1)).
      transmit(client, dev, "ok", T).
      transmit(dev, client, deliv2, T2 |
               T2 <= deadline2 + max(5d, (T - approv1) * 2)).
      transmit(client, dev, "ok", T).
      transmit(dev, client, delivf, Tf |
               Tf <= deadlinef + max(5d, (T - approv2) * 2)).
      transmit(client, dev, "ok", T).
      transmit(dev, client, "done", T).
      Success

    software (dev, client, payment, paymentdeadline, ds) =
      deliverables (dev, client, deliv1, deadline1, approv1,
                                 deliv2, deadline2, approv2,
                                 delivf, deadlinef, approvf) ||
      transmit(client, dev, payment, T | T <= paymentdeadline)
in
  software ("Me", "Client", 100000, 2004.7.1, d1, 2004.1.1, 2004.1.15,
            d2, 2004.3.1, 2004.3.15, final, 2004.7.1, 2004.7.15)
```

non-approval of a deliverable. One also sees that the contract does not specify what to do if due to delay, some approval deadline comes before the postponed delivery date. In the current code, this is taken to mean further delay on the client's part even if the client gave approval at the same time as the deliverable was transmitted. It seems that contract coding is a healthy process in the sense that it will often unveil underspecification and errors in the natural language contract being coded. The Change Order described in Section 5 of the contract and the intellectual rights described in Section 6 are not coded due to certain limitations in our language. We will postpone the discussion of this this paper's Section 6.

5 Contract Analysis

The formal groundwork in order, we can begin to ask ourselves questions about contracts such as: What is my first order of business? When is the next deadline? How much of a particular resource will I gain from my portfolio and at what times? What is the monetary value of my portfolio? Will contract fulfillment require more than the x units I currently have in stock?

The attempt to answer such questions is broadly referred to as *contract analysis*. The residuation property allows a contract analysis to be applied at any time (i.e. to any residual contract), and we can thus continuously monitor the execution of the contracts in our portfolio.

Recall that our contract specification language is parameterized over the language of predicates and arithmetic. There is a clear trade-off in play here: a sophisticated language buys expressiveness, but renders most of the analyses undecidable.

There is another source of difficulties. Variables may be bound to components of an event that is unknown at the time of analysis. An expression like $\mathrm{transmit}(a_1, a_2, R, T|\mathrm{true})$. offers little insight into the nature of R unless furnished with a probability vector over all resources.

Here we will circumvent these problems by making do with a restricted predicate language and accepting that analyses may not give answers on all input (but will give correct answers).

The predicate language is plugged in at two locations. In function application $f(\boldsymbol{a})$ where all components of the vector $\boldsymbol{a}$ must checked according to the rules of the predicate language, and in $\mathrm{transmit}(a_1, a_2, r, t|P)$ where P must have the type $\mathrm{Boolean}$. As previously we require that a_1, a_2, r, and t are either variables (bound or unbound) or constants. If some components are bound variables or constants, they must be equal to the corresponding components of an incoming event (a_1', a_2', r', t') for a match to occur.

Consider the syntax provided in figure 11. In addition to the types Agent, $\mathrm{Resource}$, and Time, the language has the fundamental types Int and $\mathrm{Boolean}$. Take τ to range over $\{\mathrm{Int}, \mathrm{Time}\}$, take σ to range over $\tau \cup \{\mathrm{Agent}, \mathrm{Resource}\}$, and assume that constants can be uniquely typed (e.g. time constants are in ISO format, and agent and resource constants are known).

The language allows arithmetic on integers, simple propositional logic, and manipulation of the two abstract types $\mathrm{Resource}$ and Time. Given a time (date) t we may add an integral number of years, months or days. For example $2004.1.1 + 3d + 1y$ yields $2005.1.4$. Resources permit a projection on a named component (field) and all fields are of type Int. E.g. to extract the total amount from an information resource named $invoice$ we write $\#(invoice, total, t)$ where t is some date[5]. The fields of resources may change over time; hence the third Time parameter.

Observables can now be understood simply as fields of a ubiquitous resource named `obs`. An Int may double for a $\mathrm{Resource}$ in which case the Int is understood to be a currency amount.

Fig. 11 Example syntax for predicate language

$$\frac{\Delta \vdash \Delta(var) = \sigma}{\Delta \vdash var : \sigma} \qquad \frac{\Delta \vdash \mathrm{type}(const) = \sigma}{\Delta \vdash const : \sigma} \qquad \frac{\Delta \vdash e_1 : \mathrm{Int} \quad \Delta \vdash e_2 : \mathrm{Int} \quad \mathrm{op} \in \{+, -, *, /\}}{\Delta \vdash e_1 \ \mathrm{op}\ e_2 : \mathrm{Int}}$$

$$\frac{\Delta \vdash t : \mathrm{Time} \quad \Delta \vdash e : \mathrm{Int} \quad \mathrm{f} \in \{\mathtt{y}, \mathtt{m}, \mathtt{d}\} \quad \mathrm{op} \in \{+, -\}}{\Delta \vdash t \ \mathrm{op}\ e\ \mathrm{f} : \mathrm{Time}} \qquad \frac{\Delta \vdash e : \mathrm{Time} \quad \mathrm{f} \in \{\mathtt{y}, \mathtt{m}, \mathtt{d}\}}{\Delta \vdash e\#\mathrm{f} : \mathrm{Int}}$$

$$\frac{\Delta \vdash r : \mathrm{Resource} \quad \Delta \vdash t : \mathrm{Time} \quad f \in fields(r)}{\Delta \vdash \#(r, f, t) : \mathrm{Int}} \qquad \frac{\Delta \vdash e : \mathrm{Int}}{\Delta \vdash e : \mathrm{Resource}}$$

$$\frac{\Delta \vdash e_1 : \tau \quad \Delta \vdash e_2 : \tau}{\Delta \vdash e_1 < e_2 : \mathrm{Boolean}} \qquad \frac{\Delta \vdash e_1 : \sigma \quad \Delta \vdash e_2 : \sigma}{\Delta \vdash e_1 = e_2 : \mathrm{Boolean}}$$

$$\frac{\Delta \vdash b_1 : \mathrm{Boolean} \quad \Delta \vdash b_2 : \mathrm{Boolean} \quad \mathrm{op} \in \{\mathtt{and}, \mathtt{or}\}}{\Delta \vdash b_1 \ \mathrm{op}\ b_2 : \mathrm{Boolean}} \qquad \frac{\Delta \vdash b : \mathrm{Boolean}}{\Delta \vdash \mathtt{not}\ b : \mathrm{Boolean}}$$

Ideally, a contract analysis can be performed *compositionally*, i.e. can be implemented by recursively evaluating subcontracts. This section contains a simple analysis with this property. Space considerations prevent a walkthrough of more involved examples, but the basic idea should be clear. We will assume for simplicity that recursively defined contracts are *guarded*. The analyses are presented using inference systems defined by induction on syntax, emphasizing the declarative and compositional nature of the analyses.

[5] When a resource is introduced into the system through a match, it must be dynamically checked that it possesses the required fields. The set of required fields can be statically determined by a routine type check annotating resources with field names à la $\{date, total, paymentdeadline\}\mathrm{Resource}$. To keep things simple we omit this type extension here.

5.1 Example: Next Point of Interest and Task List

Given a contract or a portfolio of contracts it is tremendously important for an agent to know when and how to act. To this end we demonstrate how a very simple *task list* can be compiled.

Consider the definition given in Figure 12. The function gives a structured response to reflect the decision structure (the task list) of the contract. It operates on a very simple subset of the predicate language that, however, is indicative of the bulk of temporal constraints in contracts: only interval conditions of the form $a \leq T \texttt{and} T \leq b$ with T the time variable in the enclosing transmit commitment are admitted. Such a condition is abbreviated to $[a; b]$. It is important to notice that the result of the analysis may be incomplete. A task is only added if the agents agree (i.e. `a = a1`), but if `a1` is not bound at the time of analysis, the task is simply skipped. A more elaborate dataflow analysis might reveal that in fact `a1` is always bound to `a`.

Also notice the case for application $f(\boldsymbol{a})$. We expand the body of the named contract `f` given arguments `a` but only once. This measure ensures termination of the analysis, but reduces the function's look-ahead horizon. Hence, any task or point of interest more than one recursive unfolding away is not detected. This is unlikely to have practical significance for two reasons: (1) recursively defined contracts are guarded and so a `transmit` must be matched before a new unfold can occur. This `transmit` therefore is presumably more relevant than any other `transmits` further down the line; (2) it would be grossly unidiomatic that some `transmit` t_1 was required to be matched before another `transmit` t_2, but nevertheless had a later deadline than that of t_2.

Fig. 12 Task list analysis

$$\mathrm{D}, \delta, \mathrm{a}, t \vdash \mathrm{Success} \ : \ [] \qquad \mathrm{D}, \delta, \mathrm{a}, t \vdash \mathrm{Failure} \ : \ []$$

$$\frac{\models a \neq a_1 \qquad \boldsymbol{X} = (a_1, A, R, T)}{\mathrm{D}, \delta, \mathrm{a}, t \vdash \mathrm{transmit}(\boldsymbol{X} \mid [x; y]).\, c \ : \ \mathrm{do}\ []} \qquad \frac{\models t \notin [x; y]}{\mathrm{D}, \delta, \mathrm{a}, t \vdash \mathrm{transmit}(\boldsymbol{X} \mid [x; y]).\, c \ : \ \mathrm{do}\ []}$$

$$\frac{\models a = a_1 \qquad \boldsymbol{X} = (a_1, A, R, T) \qquad t \in [x; y]}{\mathrm{D}, \delta, \mathrm{a}, t \vdash \mathrm{transmit}(\boldsymbol{X} \mid [x; y]) \ : \ \mathrm{do}\ [\mathrm{transmit}(\boldsymbol{X} \mid [x; y])]}$$

$$\frac{\mathrm{D}, \delta, \mathrm{a}, t \vdash c_1 \ : \ l_1 \quad \mathrm{D}, \delta, \mathrm{a}, t \vdash c_2 \ : \ l_2}{\mathrm{D}, \delta, \mathrm{a}, t \vdash c_1 + c_2 \ : \ \mathrm{choose}[l_1, l_2]}$$

$$\frac{\mathrm{D} \vdash c_1 \ \mathrm{nullable} \qquad \mathrm{D}, \delta, \mathrm{a}, t \vdash c_1 \ : \ l_1 \qquad \mathrm{D}, \delta, \mathrm{a}, t \vdash c_2 \ : \ l_2}{\mathrm{D}, \delta, \mathrm{a}, t \vdash c_1; c_2 \ : \ \mathrm{choose}[l_1, l_2]}$$

$$\frac{\mathrm{D} \nvdash c_1 \ \mathrm{nullable} \qquad \mathrm{D}, \delta, \mathrm{a}, t \vdash c_1 \ : \ l_1}{\mathrm{D}, \delta, \mathrm{a}, t \vdash c_1; c_2 \ : \ l_1} \qquad \frac{\mathrm{D}, \delta, \mathrm{a}, t \vdash c_1 \ : \ l_1 \qquad \mathrm{D}, \delta, \mathrm{a}, t \vdash c_2 \ : \ l_2}{\mathrm{D}, \delta, \mathrm{a}, t \vdash c_1 \parallel c_2 \ : \ l_1 \mathbin{@} l_2}$$

$$\frac{(f[\boldsymbol{X}] = c) \in \mathrm{D} \qquad \mathrm{D}, \delta, \mathrm{a}, t \vdash c \ : \ l}{\mathrm{D}, \delta, \mathrm{a}, t \vdash f(\boldsymbol{a}) \ : \ l}$$

The examples given above, in their simplicity, may be extended given knowledge of the problem domain. In particular, knowledge of or forecasting about probable event sequences may be used in a manner "orthogonal" to the coding of analyses by appropriate function calls.

Analyses that are possible to implement in this way include resource flow forecasting (supply requirements); terminability by agent; latest termination; earliest termination; and valuation, or simply put: What is the value to an agent of a given contract?

6 Discussion and Future Work

The Software Development Agreement (Figure 9) provides a good setting to observe the limitations to our approach and the ramifications of the design choices made.

The Change Order is not coded. It might be cleverly coded in the current language, again using constraints on the events passed around, but a more natural way would be using higher-order contracts, i.e. contracts taking contracts as arguments. Thus, a Change Order would simply be the passing back and forth of a contract followed by an instantiation upon agreement.

Contracts often specify certain things that are not to be done (e.g. not copying the software). Such restrictions should intersect all other outstanding contracts and limit them appropriately. A higher-order language or predicates that could guard all `transmits` of an entire subcontract might ameliorate this in a natural way.

A fuller range of language constructions that programmers are familiar with is also desirable; in the present incarnation of the contract language, several standard constructions have been left out in order to emphasize the core event model. In practice, conditionals and various sorts of lambda abstractions would make the language easier to use, though not strictly more expressive, as they can be encoded through events, albeit in a non-intuitive way. A conditional that is *not* driven by events (i.e. an if-then-else) seems to be needed for natural coding in many real-world contracts. Also, a catch-throw mechanism for unexpected events would make contracts more robust.

Conversely, certain features of the language appear to be almost too strong for the domain; the inclusion of full recursion means that contracts active for an unlimited period of time, say leases, are easy to code, but make contract analysis significantly harder. In practice, contracts running for "unlimited" time periods often have external constraints (usually local legislation) forcing the contract to be reassessed by its parties, and possibly government representatives, from time to time. Having only a restricted form of recursion that suffices for most practical applications should simplify contract analysis.

The expressivity of the contract language and indeed the feasibility of non-trivial contract analysis depends heavily on the predicate language used. Predicates restricted to the form $[a; b]$ are surely too limited, and further investigation into the required expressiveness of the predicate language is desirable.

While the language is parametrized over the predicate language used, almost all real-world applications will require some model of time and timed events to be incorporated. The current event model allows for encoding through the predicate language, but an extended set of events, with companion semantics, would make for easier contract programming; timer (or "trigger") events appear to be ubiquitous when encoding contracts.

7 Related Work

The impetus for this work comes from two directions: the REA accounting model pioneered by McCarthy [McC82] and Peyton Jones, Eber and Seward's seminal article on specification of financial contracts [JES00]. Furthermore, given that contracts specify protocols as to how parties bound by them are to interact with each other there are links to process and workflow models.

Peyton Jones, Eber and Seward [JES00] present a compositional language for specifying financial contracts. It provides a decomposition of known standard contracts such as zero coupon bonds, options, swaps, straddles, etc., into individual payment commitments that are combined declaratively using a small set of contract combinators. All contracts are two-party contracts, and the parties are implicit. The combinators (taken from [JE03], revised from [JES00]) correspond to $\mathrm{Success}, \cdot \parallel \cdot, \cdot + \cdot, \mathrm{transmit}(\cdot)$ of our language $\mathcal{C}^{\mathcal{P}}$; it has no direct counterparts to $\mathrm{Failure}, \cdot; \cdot$ nor, most importantly, recursion or iteration. On the other hand, it

provides conditionals and predicates that are applicable to arbitrary contracts, not just commitments as in $\mathcal{C}^{\mathcal{P}}$, something we have found to be worthwhile also for specifying commercial contracts.

Our contract language generalizes financial payment commitments to arbitrary transfers of resources and information, provides explicit agents and thus provides the possibility of specifying multi-party contracts.

Disregarding the structure of events and their temporal properties, $\mathcal{C}^{\mathcal{P}}$ is basically a process algebra. It corresponds to Algebra of Communicating Processes (ACP) with deadlock ($\mathrm{Failure}$), free merge ($\cdot \parallel \cdot$) and recursion, but without encapsulation [BW90]. This process algebra is also part of CSP [BHR84,Hoa85]. Note that contracts are to be thought as exclusively *reactive* processes, however: they respond to externally generated events, but do not autonomously generate them.

There are numerous timed variants of process algebras and temporal logics; see e.g. Baeten and Middelburg [BM02] for timed process algebras. Their relation to our base language is not evident at this point. This is in part because our base language is not fixed yet to accommodate expressing temporal (and other) constraints "naturally," in part because the temporal notions of timed process languages seem rather low-level and distinct from the notions we have used in contract examples.

8 Acknowledgements

This work has been partially funded by the NEXT Project, which is a collaboration between Microsoft Business Solutions, The IT University of Copenhagen and the Department of Computer Science at the University of Copenhagen (DIKU). See `http://www.itu.dk/next` for more information on NEXT.

We would like to thank Simon Peyton Jones, Jean-Marc Eber, Kasper Østerbye, and Jesper Kiehn for valuable discussions on modeling financial contracts and extending that work to commercial contracts based on the REA accounting model.

References

[AE03] Jesper Andersen and Ebbe Elsborg. Compositional specification of commercial contracts. M.S. term project, December 2003.

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

[BHR84] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *J. ACM*, 31(3):560–599, 1984.

[BM02] J.C.M. Baeten and C.A. Middelburg. *Process Algebra with Timing*. Springer, 2002.

[BW90] J.C.M. Baeten and W.P. Weijland. *Process Algebra*. Number 18 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1990.

[Ebe02] Jean-Marc Eber. Personal communication, June 2002.

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

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

[JES00] 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.

[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.